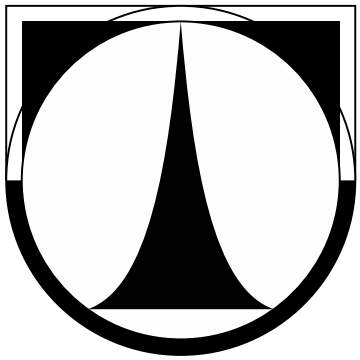
TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií



Disertační práce

Liberec 2010

Ing. Jiří Hnídek

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: P2612 Elektrotechnika a informatika
Studijní obor: 2612V045 Technická kybernetika

Síťový protokol pro grafické aplikace
Network protocol for graphics application

Ing. Jiří Hnídek

Školitel: doc. RNDr. Pavel Satrapa PhD.
Pracoviště: Ústav nových technologií a aplikované informatiky

Rozsah práce:
Počet stran: 171
Počet obrázků: 82
Počet tabulek: 4

30.12.2010

Anotace

Tato disertační práce se zabývá návrhem síťového protokolu pro realtimovou výměnu 3D dat mezi aplikacemi sdílené virtuální reality, kdy jsou na navrhovaný protokol kladeny dva protichůdné požadavky. Protokol musí zaručovat úplnou nebo částečnou spolehlivost přenášených dat. Zároveň musí protokol přenášet data s nízkými latencemi. Při návrhu protokolu se částečně vycházelo z konceptu již existujícího protokolu Verse, jehož původní návrh byl zcela přepracován s ohledem na vyšší bezpečnost, spolehlivost a efektivnost.

Práce se zabývám především návrhem nového resend mechanismu, který díky přeposílání aktuálních dat dosahuje nízkých latencí. Korektnost důležitých částí návrhu protokolu - autentizace, handshake a samotný resend mechanismus - byla ověřena programem Spin pomocí verifikačních modelů vytvořených v programovacím jazyku PROMELA.

Efektivita a spolehlivost implementace navrženého protokolu byla ověřena v experimentálním prostředí. V tomto prostředí zároveň došlo k testování vybraných transportních protokolů včetně původního protokolu Verse. Výsledky jednotlivých experimentů prokázaly, že navržený protokol může být efektivně použit pro sdílení rozsáhlých scén virtuální reality.

Klíčová slova: síťový protokol, grafická aplikace, 3D, virtuální realita

Annotation

This disertation thesis deals with the design of network protocol for exchange of 3D data between application of distributed virtual environment. Two antithetical requirements are claimed for such protocol. Protocol has to be partially or completely reliable. Neither delay jitter nor too high delay are acceptable. The draft of the protocol si partially based on the concept of existing protocol called Verse. Original draft was completly rewritten with a respect to greater security, realibility and efficiency.

The paper deals with the design of a new resend mechanism. This resend mechanism resends only actual data, thus low latency is achieved. Correctness of the important parts of the draft protocol - authentication, handshake and resend mechanism - was verified using Spin protocol. Verification models were created in the PROMELA programming language.

Efficiency and reliability of the proposed implementation of the protocol was tested in an experimental environment. Several transport protocols and the original Verse protocol were also tested in this environment. Results of experiments showed that the proposed protocol can be effectively used to share large scenes of virtual reality.

Keywords: network protocol, graphics application, 3D, virtual reality

Prohlášení

Byl jsem seznámen s tím, že na mou disertační práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé disertační práce a prohlašuji, že **souhlasím** s případným užitím mé disertační práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, že užít své disertační práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Disertační práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací se školitelem.

V Liberci 30. prosince 2010

Rád bych zde poděkoval své rodině, především Janě a Viktorce, bez jejichž podpory a trpělivosti bych tuto práci pravděpodobně nedokončil.

Dále bych zde chtěl poděkovat svému vedoucímu doc. RNDr. Pavlu Satrapovi Ph.D. za svědomité vedení mé disertační práce, Mgr. Davidovi Kmochovi za pečlivé čtení mých článků, Ing. Jirkovi Jeníčkovi Ph.D za cenné rady a Ing. Jirkovi Vranému Ph.D. za motivaci.

Obsah

Seznam obrázků

Seznam tabulek

Seznam symbolů a zkratek

ASVR Application of Shared Virtual Reality

CC Congestion Control

DCCP Datagram Congestion Control Protocol

DED Data Exchange Definition

DoS Denial of Service

DDoS Distributed Denial of Service

DTLS Datagram Transport Layer Security

ECN Explicit Congestion Notification

FC Flow Control

FPS Frames per Second

MAC Message Authentication Code

MMORPG Massively Multiplayer Online Role Playing Game

MTU Maximum Transmission Unit

NTP Network Time Protocol

PMTU Path Maximum Transmission Unit

RTT Round Trip Time

TCP Transmision Control Protocol

TLS Transport Layer Security

VR Virtual Reality

SCTP Stream Control Transport Protocol

SRTT Smoothed Round Trip Time

SVR Shared Virtual Reality

UDP User Datagram Protocol

URL Uniform Resource Locator

WFQ Weighted Fair Queuing

Kapitola 1

Úvod

V oblasti počítačové grafiky se čím dál častěji setkáváme s požadavkem na přenos dat mezi grafickými aplikacemi. V grafických studiích je to dáno tím, že se používají aplikace se specifickým zaměřením na daný úkol (fyzikální simulace, vizualizace, animace postav, apod.). Zároveň dochází k úzké profesní specializaci grafiků pracujících s těmito aplikacemi. V situaci, kdy spolupracuje několik lidí v jednom týmu nebo uživatel používá více různých aplikací pro svoji práci, tak narážíme na problém s konverzí a přenosem dat. Obecně platí, že výstupní soubor jedné aplikace je vstupem pro jinou aplikaci. Většinou se data v tomto případě ukládají do textových souborů, kdy při velkém objemu dat neúměrně narůstá velikost výsledných souborů i doba pro jejich ukládání a načítání. V pracovních postupech uživatelů se stále setkáváme s formátem OBJ Wavefront nebo nověji s formátem COLLADA [3] postaveném na technologii XML, který má ambice stát se průmyslovým standardem v dané oblasti. Obě technologie ovšem neřeší problém nezbytnosti ukládání dat do souboru a jeho opětovné načítání v jiné aplikaci i při sebemenší změně dat.

Další oblast počítačové grafiky, kdy se přenášejí data mezi aplikacemi, jsou aplikace sdílené virtuální reality (SVR). Data jsou v tomto případě přenášena pomocí síťového protokolu. Nejrozšířenějšími aplikacemi SVR jsou bezpochyby herní aplikace typu first-person shooter (FPS), které většinou nevyžadují sdílení geometrie a topologie 3D objektů. Vystačí si se sdílením polohy jednotlivých avatarů a jejich stavů. Pokud se uživatelé ASVR mají setkávat a společně měnit prostředí virtuální reality, tak se protokoly herních aplikací jeví jako nedostatečné. V takovém případě je potřeba odlišný přístup návrhu síťového protokolu. Nároky na něj jsou ovšem velmi protichůdné. Jednak jsou vyžadovány nízké latence přenášených dat, jak to poskytuje například transportní protokol UDP. Zároveň je potřeba zaručit spolehlivost přenášených dat, ale nikoliv úplnou spolehlivost přenášených dat doručených

ve správném pořadí, tak jak to zaručuje například transportní protokol TCP.

Síťový protokol, který měl ambice být univerzálním síťovým protokolem pro komunikaci mezi grafickými aplikacemi, je protokol Verse. Verse byl od počátku navrhován speciálně pro efektivní real-timové sdílení dat. Na jeho vývoji se podílelo Uni-Verse konsorcium v rámci 6. rámcového program Evropské unie. Členy konsorcia bylo několik významných evropských univerzit a výzkumných institucí (KTH, Fraunhoufer Institut, Helsinky University of Technology, Interactive Institute, Blender Foundation, a další). Tento projekt měl za cíl i vývoj podpůrného aplikačního vybavení nebo integraci protokolu do vybraných grafických aplikací. Bohužel se možná až příliš zaměřil spíše na vývoj aplikací používající protokol Verse než na vývoj samotného protokolu. Po ukončení financování z Evropské unie vývoj protokolu Verse bohužel víceméně ustal a protokol Verse se nakonec z mnoha důvodů příliš nerozšířil. Cílem této disertace bylo navrhnout lepší náhradu původního protokolu.

Práce je organizována tímto způsobem: teoretický úvod do návrhu síťových protokolů. Další kapitola je věnována popisu původního protokolu Verse, na kterou navazuje kapitola věnovaná návrhu nového protokolu Verse. Následující kapitola se věnuje doporučením, jak provádět implementaci nového protokolu. Poslední kapitola obsahuje výsledky výkonnostních testů.

Kapitola 2

Úvod do teorie návrhu síťových protokolů

Návrh síťového protokolu je netriviální činnost při které není vhodné se spoléhat pouze na vlastní úsudek jako je to možné při návrhu sekvenčního algoritmu. Při návrhu síťového protokolu je nutné uvažovat, že spolu komunikují dva a více počítačů. Pokud autor navrhuje složitější síťový protokol, tak není v jeho silách uvažovat všechny možné scénáře komunikace. Už při samotné tvorbě návrhu se může dopustit mnoha chyb, které ve výsledku vedou k nepředvídatelnému chování protokolu.

Při návrhu síťového protokolu je vhodné nejprve vytvořit detailní specifikaci síťového protokolu. Zároveň je vhodné, aby tvůrci návrh protokolu formalizoval do zjednodušeného modelu a provedli verifikaci tohoto modelu. Při ověřování návrhu protokolu je možné použít celou řadu programových nástrojů, které umožňují odhalit jeho chyby. Jedním z takových nástrojů je program Spin [21], který slouží k verifikaci systémů popsaných pomocí programovacího jazyka PROMELA. Program Spin umožňuje provádět simulaci vytvořeného modelu, nebo umožňuje vygenerovat program v jazyce C, který může provést úplnou prohlídku stavového prostoru daného systému. Spin může během prohlídky stavového prostoru provádět kontrolu existence uváznutí, nevyvíjejících se cyklů, nekorektních koncových stavů, apod. Další neméně důležitou vlastností je možnost ověřovat vlastnost systému pomocí lineární temporální logiky (LTL).

Po verifikaci návrhu protokolu následuje jeho implementace ve vhodném programovacím jazyku. Vlastní implementaci je vhodné otestovat a následně nasadit do reálného provozu. V ideálním případě už není nikdy potřeba protokol měnit, protože velmi dobře škáluje a je možný vyměnit kterýkoliv protokol na nižší vrstvě. V praxi se při testování nebo jeho nasazení mohou objevit potíže, které autoři ve svém původním návrhu neočekávali či nepředvídali,

což často vede k nutnosti změnit původní specifikaci. Změna ve specifikaci by v ideálním případě měla být opět následována verifikacím změněného modelu a následně implementací a testováním. Návrh síťového protokolu je živý proces, který nemusí nikdy ustat. Příkladem protokolu, který se stále vyvíjí a snaží se reflektovat vývoj síťových technologií, je například protokol TCP.

Proti snahám o změnu specifikace původního protokolu jde naopak potřeba o zachování zpětné kompatibility s původní specifikací. Síťové prvky, koncová zařízení a aplikace očekávají nebo vyžadují určité chování protokolu a jeho časté a zásadní změny může mít negativní dopad na fungování aplikací nebo celé sítě.

První, nejdůležitější částí návrhu protokolu je tedy vytvoření jeho specifikace. Každá specifikace protokolu by měla obsahovat pět částí.

1. Definice a popis služby, kterou bude protokol poskytovat
2. Předpoklady o prostředí, ve kterém bude protokol provozován
3. Definice zpráv, které se budou používat k implementaci protokolu
4. Formát a způsob kódování zpráv
5. Seznam pravidel, které budou zajišťovat správné doručení dat

Při popisu prostředí, ve kterém bude protokol provozován, se často využívá tzv. vícevrstvý model, který velmi zjednodušuje návrh většiny protokolů. Ve specifikaci je často uvedeno, na jaké vrstvě je daný protokol zamýšlen a jaké požadavky jsou kladeny na nižší vrstvu. V mnoha případech je v návrhu protokolu uvedeno, že na nižší vrstvě je vyžadován nějaký konkrétní protokol. Při definici zpráv je málokdy možné použít výčet všech možných zpráv. Většina protokolů má tudíž nějakou hierarchickou strukturu zprávy jako je například hlavička, tělo zprávy, apod. Specifikace pak obsahuje popis struktury zpráv a její možné hodnoty. Nejtěžší částí návrhu protokolu je vytvoření seznamu pravidel, která zajistí jeho správné fungování.

Při vlastním návrhu specifikace protokolu a následné implementaci je vhodné dodržovat určité zásady. G. J. Holzman [21] navrhuje při návrhu protokolu dodržovat následujících 10 kroků.

1. Nejprve je vhodné ujistit se, že problém je dobře definovaný. Před vlastním návrhem musí být jasná a zřejmá všechna kritéria, požadavky a omezení, která budou na navrhovaný protokol kladena.
2. Dále je vhodné definovat službu, kterou bude daný protokol poskytovat.

3. Před návrhem vnitřní funkcionality programu je výhodné navrhnout jeho vnější funkcionalitu, neboli jak bude protokol komunikovat s nadřazenou vrstvou. V tomto kroku návrhu je vhodné o protokolu uvažovat jako o černé skříňce, která splňuje požadavky z předchozích kroků.

4. Návrh protokolu by měl být co možná nejjednodušší, protože komplikované protokoly jsou náchylnější na chyby, složitě se implementují, verifikují a často nejsou tak efektivní jako jednoduché protokoly. Problém, který se jeví jako komplikovaný, lze často rozložit na několik jednoduchých podproblémů a řešit je samostatně.

5. Není dobré spojovat, co je nezávislé.

6. Dobře navržený protokol nezakazuje irelevantní a nedůležité věci. Dobře navržený protokol by měl mít možnost se dále vyvíjet.

7. Před vlastní implementací protokolu je vhodné vytvořit model protokolu a ověřit, že jsou splněna všechna kritéria návrhu protokolu.

8. Teprve v tomto kroku dojde k vlastní implementaci protokolu, výkonnostním testům a případné optimalizaci kódu.

9. Po implementaci by mělo být ověřeno, že se implementace chová stejně jako model vytvořený v 7. kroku.

10. Není vhodné vynechat 1. až 7. krok.

Výše uvedená pravidla nejsou nařízením, jak postupovat při návrhu protokolu, ale mají sloužit pouze jako doporučení pro autory protokolu. Nutno podotknout, že se při návrhu síťových protokolů nejčastěji porušuje 10. pravidlo

Kapitola 3

Původní protokol Verse

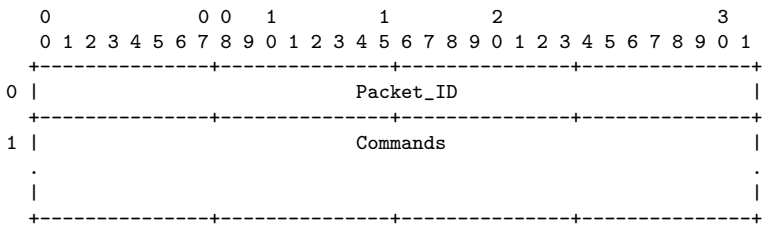
Vývoj původního protokolu Verse započal Eskil Steenberg a Emil Brink v roce 1999 na KTH. Tento protokol byl navrhován poměrně živelně a jeho návrh se rozhodně neprováděl podle zásad popsaných v druhé kapitole. V konečném důsledku má specifikace [32] i implementace původního protokolu Verse mnoho nedostatků. Na druhou stranu v jeho specifikaci lze nalézt několik zajímavých myšlenek, které byly inspirací pro nový návrh specifikace protokolu Verse. Aby bylo možné porovnat původní a nový návrh, tak v této kapitole budou popsány základní vlastnosti původního protokolu Verse.

Protokol Verse používá architekturu klient-server. Na transportní vrstvě vyžaduje protokol UDP, protože umožňuje zaručit nízkou latenci doručených dat a je široce podporovaný. Protokol poskytuje částečně spolehlivý přenosový kanál, který přeposílá pouze aktuální data. Protokol byl od počátku navržen speciálně pro real-timové sdílení 3D dat na serveru. Pokud klient provede změnu ve sdílených datech (poloha objektu, topologie objektu, apod.), tak by měl o této změně odeslat příslušnou zprávu serveru. Server se následně musí postarat o přeposlání zprávy všem klientům, kteří se o tato sdílená data zajímají.

3.1 Struktura paketu

Každý paket původního protokolu má velice jednoduchou strukturu:

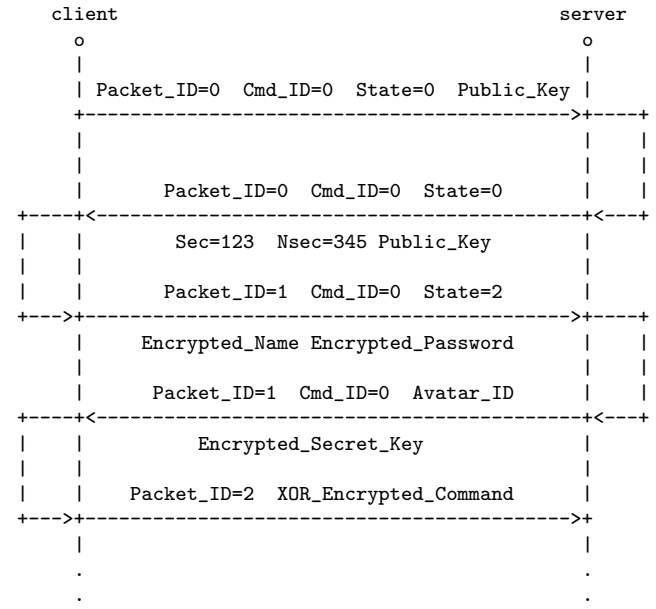
Hlavička obsahuje pouze Packet_ID, který slouží jako je jedinečný identifikátor paketu, který odesílající strana inkrementuje s každým odeslaným paketem. Za hlavičkou následují příkazy, jejichž typ, délka a struktura je pevně daná prvním bytem každého příkazu. Taková jednoduchá struktura paketu má výhodu v tom, že se šetří místem na důležitá data, ale na druhou stranu není příliš flexibilní.



Obrázek 3.1: Struktura paketu původního protokolu Verse

3.2 Handshake

Handshake mezi klientem a serverem má následující podobu:



Obrázek 3.2: Příklad výměny paketů během handshaku

Klient zahajuje spojení se serverem zasláním paketu, kde se server dozví RSA veřejný klíč klienta. Server odpovídá zasláním paketu, který obsahuje kromě RSA veřejného klíče serveru také časovou značku, která slouží k synchronizaci času na straně klienta a serveru. Když klient zná veřejný klíč serveru, tak pomocí něho zašifruje jméno a heslo a pošle ho serveru. Server zašifrované jméno a heslo rozšifruje pomocí tajného klíče a pokud se shoduje s jeho záznamy, tak na tento paket odpoví paketem obsahujícím příkaz potvrzující připojení. Tento příkaz zároveň obsahuje jedinečný identifikátor avatara a tajný klíč pro symetrickou šifru, který je zašifrovaný pomocí

veřejného klíče klienta. Klient může po obdržení tohoto paketu začít posílat serveru pakety zašifrované pomocí symetrické šifry. V každém kroku handshaku posílá klient svůj požadavek opakovaně, dokud nedostane od serveru odpověď nebo dokud nevyprší timout pro dané spojení, který je pro každé spojení 30 sekund.

Používaný handshake je zranitelný proti Man-in-the-middle útoku, protože navržený handshake neobsahuje žádný mechanismus, který by mohl ověřit identitu serveru. Jediným možným způsobem jak se bránit proti man-in-the-middle útoku je přenesení veřejného klíče serveru nějakým jiným zabezpečeným kanálem a při handshaku kontrolovat shodu těchto klíčů. Další nedostatek spočívá v zaslání pouze jedné časové značky jako odpověď na první paket. Ke zjištění přesného času na serveru je toto naprosto nedostatečný mechanismus. Bylo by vhodnější zavést mechanismus používaný v protokolu NTP.

Dalším vážným nedostatkem je použitá symetrická šifra. Šifrování i dešifrování se provádí pomocí velmi jednoduchého algoritmu, který je popsán pomocí následujícího pseudo-kódu:

```
begin
  pos := packet_id;
  pos := key[pos  mod key_size];
  for i := 0 to data_size step 1 do
      data[i] := data[i] xor key[(i + pos)  mod key_size];
  od
end
```

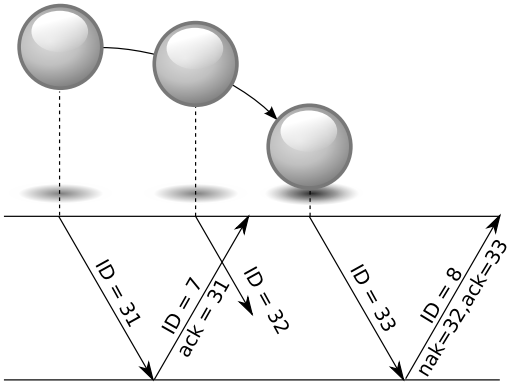
Když vezmeme v úvahu, že velikost tajného klíče je pevně nastavena na 64 bytů a že k šifrování se používá de-facto pouze funkce XOR, tak jde o velmi chatrné zabezpečení. Situaci navíc zhoršuje fakt, že se nepoužívá žádné vyplňování náhodnými čísly a že většina klientů posílá na začátku spojení podobnou sekvenci příkazů.

3.3 Resend mechanismus

Původní protokol Verse používá velmi zajímavý koncept resend mechanismu, který se snaží přeposlat pouze ta data, která jsou aktuální. Navíc je detekce ztráty paketu navržena tak, aby docházelo k co možná nejmenšímu zpožděním při přeposílání paketu. Když bude uvažován příklad z obrázku 3.3, tak ztráta paketu 32 není detekována odesílající stranou na základě vypršení

časovače daného paketu, protože takový přístup by vedl ke zbytečně velkým zpožděním.

Použitý přístup bude popsán na následujícím zjednodušeném příkladu, kdy odesílatel posílá v pravidelných intervalech polohu objektu. Když příjemce úspěšně přijme paket 31, tak přijetí tohoto paketu potvrdí odesláním paketu obsahujícího příkaz $ACK = 31$ a zároveň očekává přijetí paketu 32. Ten je ovšem ztracen. Příjemce detekuje ztrátu paketu 32 tehdy, když přijme paket 33. Paket 32 je považován za definitivně ztracený, protože autoři původního protokolu uvažovali, že na síti dochází změně pořadí paketů pouze ve speciálních případech, kdy síť nefunguje korektně. Samozřejmě i v tomto existují vyjímky.



Obrázek 3.3: Zjednodušený příklad výměny paketů

Příjemce potvrdí přijetí paketu 33 a ztrátu paketu 32 odesláním paketu obsahujícího příkazy $NAK = 32$ a $ACK = 33$. Když odesílatel obdrží tento paket, tak nedojde k prostému přeposlání paketu 32, protože by se přeposlala neaktuální pozice objektu. Odesílatel musí z paketu 32 přeposlat pouze ta data, která již nebyla odeslána v nějakém paketu, jehož $Packet\_ID$ bylo větší jak 32.

Výše popsaný resend mechanismus má výhodu v tom, že není nutné přeposílat neaktuální data a detekce ztráty paketu je poměrně efektivní, protože ke ztrátě paketu dochází většinou tehdy, když se odesílá velké množství paketů a dojde k zahlcení přenosových cest. Na druhou stranu výše uvedený koncept má několik nedostatků. K potvrzení přijetí paketu může dojít pouze tehdy, když příjemce posílá paket. K tomu může dojít ze třech důvodů. Příjemce má nějaká data, která chce přeposlat protistraně. Příjemce musí poslat tzv. keep-alive packet, který se posílá každé dvě vteřiny. Posledním důvodem je nutnost urgentně doručit informaci o ztrátě paketu.

Dalším nedostatkem resend mechanismu je fakt, že se s potvrzovacími příkazy ACK a NAK zachází jako s jakýmkoliv jinými příkazy. Jinými slovy ACK a NAK příkazy jsou zaslány odesílateli jenom jednou a jejich případná ztráta se detekuje opět na straně odesílatele.

Posledním nedostatkem je absence jakákoliv komprese ACK a NAK příkazu, takže při velkém zpoždění mezi odesílatelem a příjemce dochází k vyplnění velké části paketu pouze ACK a NAK příkazy.

3.4 Datový model

Jedním z cílů protokolu Verse bylo vytvořit síťový protokol který by umožňoval komunikaci rozdílných aplikací. Aby bylo možné sdílet mezi aplikacemi data, byl vytvořen datový model a odpovídající sada zpráv pro výměnu těchto dat. Datový model byl na jednu stranu navržen poměrně obecně, ale některé části obsahují zcela nelogická a zbytečná omezení.

Všechna data jsou uložena v tzv. uzlech. Každý uzel má svůj jedinečný identifikátor a typ, který určuje jeho další možnou strukturu. Rozlišuje se 7 typů uzlů:

1. Objektový uzel
2. Geometrický uzel
3. Materiálový uzel
4. Bitmapový uzel
5. Textový uzel
6. Křivkový uzel
7. Audio uzel

Objektový uzel umožňuje sdílet informaci o poloze, rotaci a velikosti objektu. Zároveň může tento uzel sloužit jako zdroj světla a může být propojen s jiným uzlem. Každý uzel umožňuje sdílet informace v tzv. tagech a skupinách tagů. Tag má v rámci své skupiny jedinečný identifikátor a jméno, které umožňují jeho adresování v příkazech. Tag může nést číselnou informaci, logickou hodnotu nebo řetězec. Další typu uzlů jsou popsány ve specifikaci [32] nebo v článku [9].

Aby se Verse klient dozvěděl o uzlech sdílených na serveru, tak musí pomocí speciálního příkazu *node_index_subscribe* zažádat o přihlášení k danému

typu uzlů. Verse server na tento požadavek odpovídá zasláním sady příkazů *node_create*. Klient se následně může přihlásit k vlastním datům daného uzlu.

Datový model na jednu stranu tlačí vývojáře, aby například sdíleli transformaci objektů v objektovém uzlu, geometrii a topologii sítě v geometrickém uzlu, ale na druhou stranu neumožňuje sdílet informace o hranách. Neumožňuje sdílet informace o ploškách, které obsahují více jak 4 hrany, atd. Výsledný datový model i celý síťový protokol je dosti těžkopádný, neflexibilní. Nehledě na to, že jeho implementace v dalším programovacím jazyku by byla dosti pracná.

Kapitola 4

Nový protokol Verse

Rozhodnutí vytvořit zcela nový protokol mělo mnoho důvodů. Jeho původní ad-hoc návrh, způsob implementace a nekompletní specifikace neumožňovaly návrh změnit, aby výsledný protokol byl robustní, spolehlivý a zároveň zpětně kompatibilní. Dalším důvodem pro návrh nového protokolu byla zkušenost s implementací starého protokolu do programu 3D modelovacího a animačního programu Blender, která byla velmi komplikovaná a v konečném důsledku nekompletní, pomalá a nestabilní. Výsledná implementace byla prezentována na konferenci BCONF 2005 [17].

Nový protokol Verse byl zcela přepracován a při jeho tvorbě se postupovalo podle zásad popsaných v druhé kapitole. Zároveň byl brán zřetel na to, aby jeho budoucí implementace a integrace do existujících i nových aplikací byla co možná nejjednodušší.

4.1 Požadavky na nový protokol

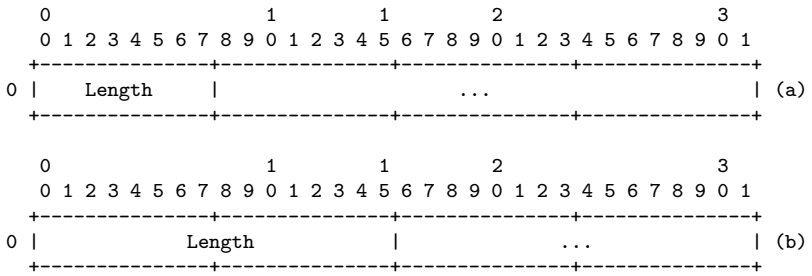
Všechny podmínky na navrhovaný protokol vycházejí z myšlenky, že protokol by měl umožňovat v reálném čase sdílet data mezi grafickými aplikacemi pomocí sítě Internet. Z tohoto lze odvodit spoustu dílčích požadavků:

- Data musí být přenášena s minimálními latencemi
- Pro přenos dat je vyžadována částečná spolehlivost
- Protokol musí mít mechanismus pro navázání spojení a přátelské ukončení spojení
- Bude kladen velký důraz na zabezpečení celého protokolu
- Bude možné dohadovat vlastnosti spojení

- Protokol bude implementovat vlastní sadu Congestion Control mechanismů
- Protokol bude podporovat obecný datový model
- Přístup k sdíleným datům bude možné omezit pomocí přístupových práv

4.2 Kódování dat

Všechny vícebytové celočíselné hodnoty jsou přenášeny jako big-endian (nejvíce významný byte je první). Protokol umožňuje přenášet i číselné hodnoty v plovoucí desetinné čárce. Konkrétně jsou podporovány formáty s jednoduchou (real32) a dvojitou přesností (real64) jak je definuje standard IEEE 745. Pro přenos řetězců jsou definovány dvě struktury *strin8* a *string16* jež jsou uvedeny na obrázku 4.1.



Obrázek 4.1: Struktury string8 (a) a string16 (b) pro přenos řetězců

První položka obsahuje vždy délku řetězce v bytech. Položka *Length* se nepočítá do délky řetězce. Řetězec by neměl být ukončen znakem 0x00, jak to vyžaduje například programovací jazyk C/C++. Takový znak by měl být z řetězce odstraněn. Pokud není řečeno jinak, tak řetězec by měl být kódován pomocí UTF-8 [37].

4.3 Transportní protokol

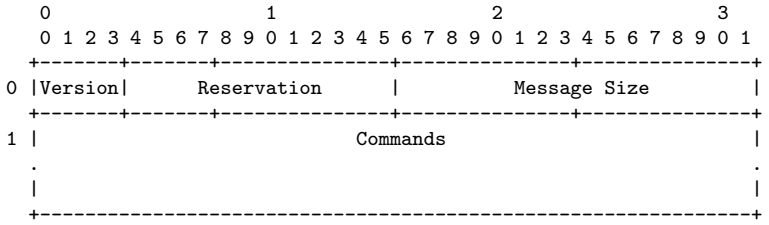
Před rozhodnutím jaký použít transportní protokol byla provedena sada testů s cílem porovnat vhodnost jednotlivých transportních protokolů pro aplikace sdílené virtuální reality (ASVR). Výsledky experimentů jsou uvedeny na straně 101.

Výsledky měření ukázaly, že na transportní vrstvě by nebylo vhodné nasazovat TCP ani spolehlivou variantu SCTP, protože tyto protokoly vedou při real-timovému sdílení dat k příliš vysokým latencím. Při použití částečné spolehlivé varianty protokolu SCTP by se sice předešlo velkým latencím, ale timout omezující přeposílání by komplikoval návrh vlastního resend mechanismu, který má zajišťovat specifické vlastnosti protokolu Verse. DCCP se může zdát jako vhodný kandidát, ale má několik vlastností, které by vedly k neefektivnímu využívání přenosových cest. V jeho neprospěch hraje i fakt, že není široce podporován na koncových zařízeních ani síťových prvcích. Jeho možné budoucí nasazení na transportní vrstvě se ovšem úplně nevylučuje. Jako nejvhodnějším kandidátem se nakonec ukázal opět protokol UDP, protože umožňuje přenos s nízkými latencemi a zároveň je široce rozšířený. Při dalším návrhu se ovšem uvažovalo i s alternativou, že na transportní vrstvě může být v budoucnu použit i jiný vhodný datagramový transportní protokol než je UDP.

Přestože se protokol TCP ukázal jako nevhodný pro real-timový přenos dat, v návrhu protokolu se s ním počítá pro zahájení spojení. Na zabezpečeném TCP spojení nejprve proběhne autentizace uživatele a následně si klient se serverem dohodnout typ datagramového transportního protokolu a další vlastnosti spojení pro real-timové sdílení dat. Použití TCP protokolu pro navázání spojení může mít i tu výhodu, že pokud není možné dohodnout použití UDP, protože to například nedovoluje mobilní operátor, tak spojení pro real-timovou výměnu dat může být degradováno na původní TCP spojení. Dále může být teoreticky dvojice TCP a UDP spojení použita pro efektivní streamování velkého 3D datového souboru, jak to ukázali Al-Regib a Altunbasak ve svém článku [1].

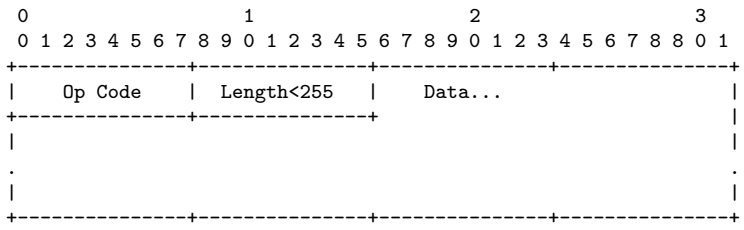
4.4 Struktura zpráv a příkazů

Na TCP spojení jsou data přenášena pomocí zpráv, jež mají strukturu popsanou na obrázku 4.2.



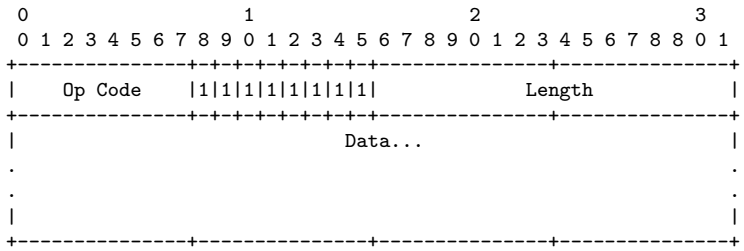
Obrázek 4.2: Struktura zprávy posílaná přes zabezpečené TCP spojení

Každá zpráva má jednoduchou hlavičku, která obsahuje *Version*: verzi protokolu, *Reservation*: rezervaci pro další položky a *Message Size*: velikost zprávy včetně hlavičky v bytech. Za hlavičkou následují příkazy, které mají strukturu popsanou na obrázku 4.3.



Obrázek 4.3: Struktura krátké varianty příkazu

Každý typ příkazu má svůj jedinečný *Op Code*, který specifikuje další strukturu příkazu. Za položkou *Op Code* následuje *Length*: délka příkazu (zahrnující i položky *Op Code* a *Length*), která je udávána v bytech a může nabývat hodnot v rozsahu $\langle 2, 254 \rangle$. Pokud je nutné poslat příkaz delší jak 254 bytů, má strukturu popsanou na obrázku 4.4



Obrázek 4.4: Struktura dlouhé varianty příkazu

V tomto příkazu může být přeneseno až 65531 bytů dat. Položka s délkou příkazu není záměrně volena větší. Hlavním důvodem je maximální velikost zprávy, která je 65535 bytů. Navíc stejná struktura příkazu se používá i pro real-timový přenos dat a UDP ani DCCP neumožňují přenést v jednom datagramu více jak 65535 bytů.

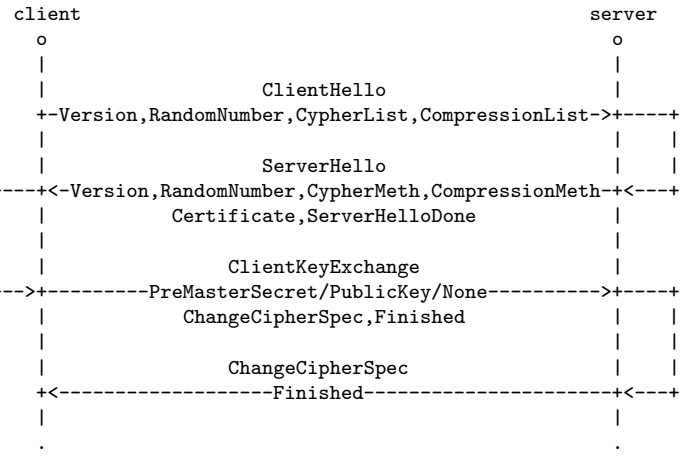
4.5 Navázání spojení

Komunikaci zahajuje klient na TCP spojení, které musí být šifrované pomocí protokolu Transport Layer Security (TLS) [11], protože během této části jsou přenášeny citlivé informace jako je například uživatelské jméno a heslo,

které nesmí být odposlechnuty útočníkem. Příklad kompletního handshaku je uveden na obrázku 4.21, který je poměrně komplikovaný. Dá se ovšem rozdělit na několika částí.

4.5.1 TLS handshake

První částí navázání spojení je TLS handshake, který je detailně popsaný v příslušném RFC dokumentu [11]. Zjednodušený průběh TLS handshaku je vidět na obrázku 4.5. V této části budou zmíněny základní mechanismy a vlastnosti protokolu TLS.



Obrázek 4.5: Zjednodušený průběh TLS handshaku

TLS je aplikační protokol využívající na nižší vrstvě spolehlivý transportní protokol (např. TCP). Spojení se serverem zahajuje klient zasláním zprávy *ClientHello*, která obsahuje verzi protokolu, náhodně vygenerované číslo, seznam navrhovaných šifer a seznam kompresních metod. Server na tuto zprávu odpoví zprávou *ServerHello*, která obsahuje stejnou verzi protokolu, jakou se rozhodl použít klient. Dále tato zpráva obsahuje náhodně vygenerované číslo, vybranou sadu šifer a kompresní metodu. Server zároveň zašle klientovi svůj certifikát. Server může zaslat klientovi i zprávu *CertificateRequest*, kterou vyžaduje po klientovi zaslání jeho vlastního certifikátu. Toto ovšem v případě protokolu Verse není vyžadováno. Server první část komunikace ukončuje zprávou *ServerHelloDone*.

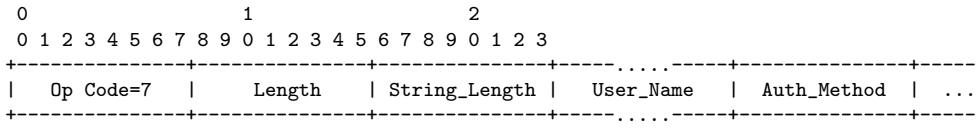
Klient z obdrženého certifikátu získá veřejný klíč serveru a digitální podpis certifikátu. Klient by měl ověřit platnost veřejného klíče z certifikátu vydavatele, který musí být předem nahraný na straně klienta. Po ověření platnosti veřejného klíče, ověří klient digitální podpis v obdrženém certi-

fikátu. Certifikát by měl být samozřejmě platný a měla by se shodovat adresa serveru uvedená v certifikátu s adresou serveru, který certifikát odeslal.

Když klient ověří identitu serveru, tak odešle serveru zprávu *ClientKeyExchange*, která může obsahovat v závislosti na zvolené symetrické šifře *PreMasterSecret*, *veřejný klíč* anebo *nic*. Klient i server si z obsahu *ClientKeyExchange* a náhodných čísel spočítají tzv. master secret z kterého jsou odvozeny všechny ostatní klíče. Následně klient odešle zprávu *ChangeCipherSpec*, kterou říká, že následující data jsou šifrovaná. Tuto část handshaku klient zakončí zašifrovanou zprávou *Finished*, jež obsahuje hash a MAC všech předchozích zpráv. Server se tuto zprávu pokusí dešifrovat a ověřit hash a MAC. Pokud dešifrování a ověření neselže, tak server odpoví obdobnými zprávami *ChangeCipherSpec* a *Finished*. V případě, že i klient je schopný dešifrovat zprávu *Finished* a ověřit hash a MAC, tak je TLS handshake ukončen a obě strany mohou zahájit komunikaci po zabezpečeném spojení.

4.5.2 Autentizace

Po ukončení TLS handshaku začíná vlastní komunikace pomocí protokolu Verse a klient 30 sekund na to, aby poslal zprávu obsahující příkaz *UserAuthRequest*. Jeho struktura je popsána na obrázku 4.6. Pokud server tento příkaz do 30 sekund neobdrží, server by měl spojení s klientem automaticky ukončit.



Obrázek 4.6: Struktura příkazu UserAauthRequest

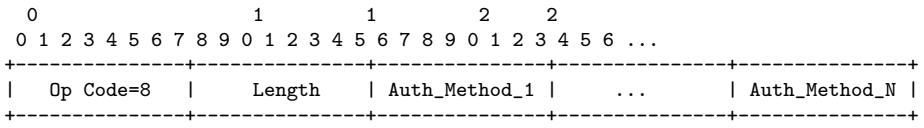
Op Code tohoto příkazu je 7. Za délkou příkazu následuje délka řetězce v němž je uloženo uživatelské jméno. Za uživatelským jménem následuje identifikátor autentizační metody. Celý příkaz je ukončen vlastními autentizačními daty. Seznam aktuálně navrhovaných autentizačních metod je uveden v tabulce 4.1. Celý autentizační mechanismus byl navržen flexibilně, aby bylo možné v budoucnu přidat další autentizační metody.

Přestože není doporučeno, aby server podporoval ověřování uživatelů pomocí metody NONE, klient by měl poslat první příkaz *UserAuthRequest* právě s touto metodou. Server musí odpovědět na první příkaz *UserAuthRequest* obsahující nepodporovanou metodu příkazem *UserAuthFailure* se seznamem autentizačních metod, které server podporuje. Tento seznam musí

RESERVED=0	Tato metoda by neměla být nikdy použita
NONE=1	Není doporučeno podporovat tuto metodu
PASSWORD=3	Tato metoda je vyžadována
HOSTBASED=4	Tato metoda je volitelná

Tabulka 4.1: Metody autentizace

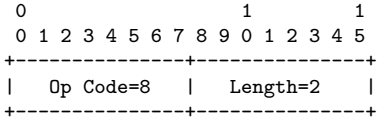
vždy obsahovat metodu PASSWORD a jak server tak klient musí tuto metodu podporovat. Klient by si měl ze seznamu obdržených metod jednu vybrat a použít ji k odeslání platné kombinace uživatelského jména a autentizačních dat. Příkaz *UserAuthFailure* je uveden na obrázku 4.7.



Obrázek 4.7: Struktura příkazu UserAauthFailure se seznamem podporovaných metod

Když klient pošle neplatnou kombinaci uživatelského jména a autentizačních dat, tak záleží na tom, kolik neúspěšných příkazů *UserAuthRequest* již klient poslal. Pokud tento počet překročil stanovený limit, tak by měl server odpovědět příkazem *UserAuthFailure* uvedeným na obrázku 4.8. V opačném případě by měl server odpovědět příkazem z obrázku 4.7 a umožnit klientovi další přihlašovací pokus. Zprávu s tímto příkazem by server neměl posílat ihned, ale odpověď by měl pozdržet. Časový rozestup by měl s každým neplatným pokusem narůstat, aby se co nejvíce zkomplikovalo hádání kombinací uživatelských jmen a autentizačních dat hrubou silou. Dále server nesmí během jednoho spojení měnit seznam navrhovaných autentizačních metod ani jejich pořadí.

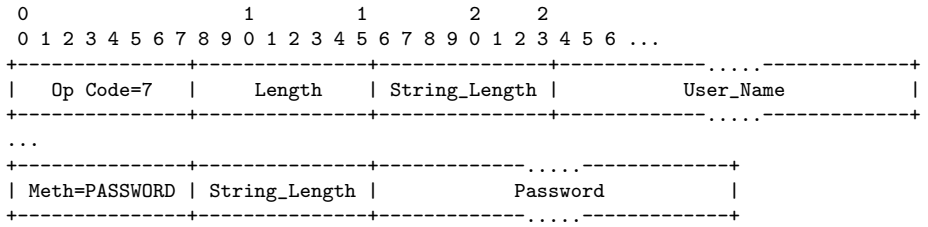
Server by měl po odeslání ukončujícího příkazu *UserAuthFailure* ukončit i TLS a TCP spojení. Server by měl spojení automaticky ukončit i v tom případě, že neobdrží další příkaz *UserAuthRequest* do 30 sekund po odeslání příkazu *UserAuthFailure* s navrhovanými metodami.



Obrázek 4.8: Struktura příkazu UserAauthFailure ukončující spojení

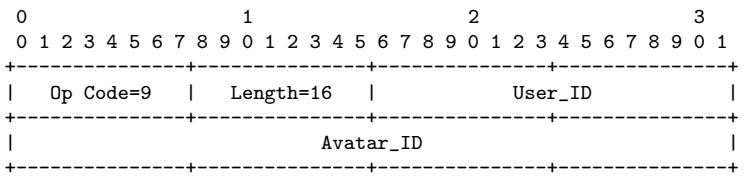
Pokud server obdrží od klienta první příkaz *UserAuthRequest* obsahující uživatelské jméno, jež není obsaženo v jeho databázi uživatelů, tak by server neměl odpovídat příkazem z obrázku 4.8, protože útočník zjišťující platné kombinace uživatelských jmen a autentizačních dat by měl ulehčenou práci. Takový přístup by umožňoval případnému útočníkovi nejprve zjistit platná uživatelská jména a tím výrazně ulehčil zjištění platné kombinace uživatelského jména a autentizačních dat.

Zatím jedinou navrhovanou autentizační metodou, která umožňuje ověřit uživatele, je metoda PASSWORD. V této metodě se uživatel prokazuje heslem, které by měl znát pouze on sám. Na serveru by toto heslo mělo být uloženo v nějaké zašifrované podobě. Struktura příkazu *UserAuthRequest* obsahující uživatelské jméno a heslo je uvedena na obrázku 4.9.



Obrázek 4.9: Struktura příkazu UserAauthRequest obsahující heslo

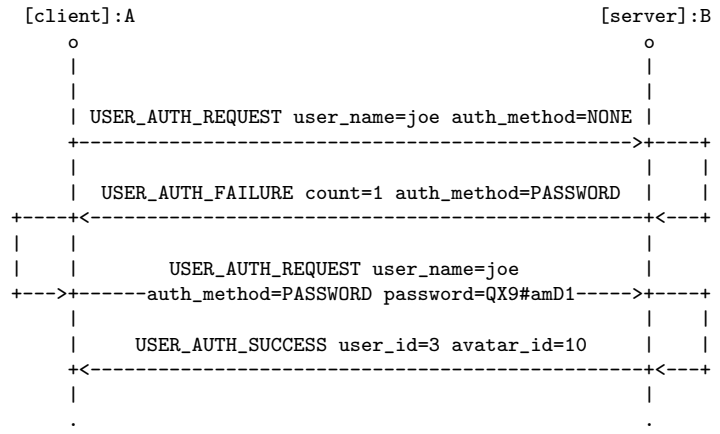
Když se uživatelské jméno a autentizační data (heslo) shodují s databází na straně serveru, server odešle klientu zprávu s příkazem *UserAuthSuccess* jehož struktura je uvedena na obrázku 4.10.



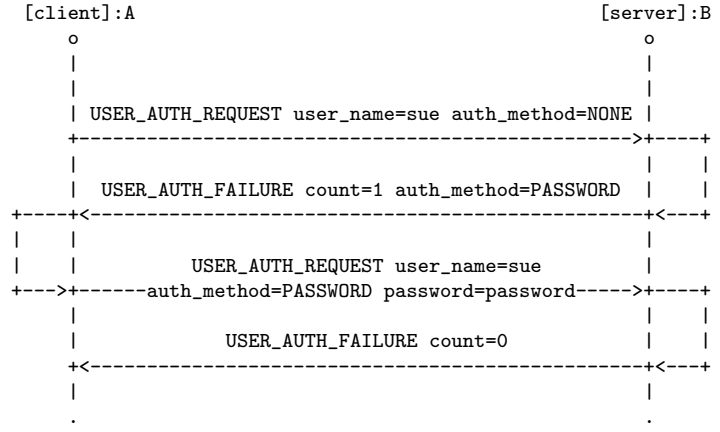
Obrázek 4.10: Struktura příkazu UserAauthSuccess

Tento příkaz obsahuje jedinečný identifikátor uživatele: *User_ID* a avatara *Avatar_ID*. *User_ID* se používá například pro nastavení vlastnictví a přístupových práv ke sdíleným datům. Po úspěšném ověření uživatele server vytvoří speciální uzel, který reprezentuje avatara daného Verse klienta. Identifikátor tohoto uzlu je roven *Avatar_ID* z příkazu *UserAuthSuccess*. Uzly reprezentující avatary budou detailně popsány v kapitole 5.

Příklad výměny zpráv a příkazů během úspěšné a neúspěšné autentizace je uveden na obrázcích 4.11 a 4.12.



Obrázek 4.11: Příklad úspěšné autentizace

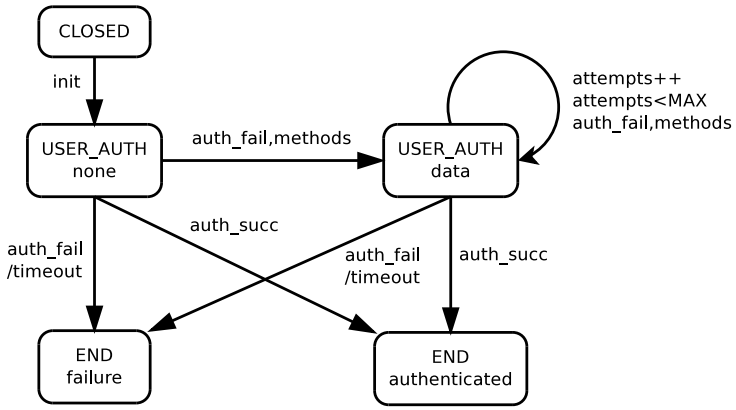


Obrázek 4.12: Příklad neúspěšné autentizace

Verifikace autentizace

Autentizační část handshaku bylo nutné verifikovat, protože je relativně komplikovaná a nezbytná pro následnou komunikaci mezi klientem a serverem. Nejprve byly vytvořeny stavové modely autentizace pro klienta a server, které jsou uvedeny na obrázcích 4.13, 4.14.

Z těchto dvou stavových modelů byl vytvořen verifikační program v programovacím jazyku Promela, který je uveden v příloze na straně 124. Při tvorbě stavových modelů byl původní formát předávaných zpráv a příkazů velmi zjednodušen. Proces reprezentující klienta může posílat skrz synchronní kanál pouze zjednodušený příkaz *UserAuthRequest*. Položky username a data mohou nabývat pouze hodnoty platná nebo neplatná data. Toto může působit



Obrázek 4.13: Stavový model autentizace na straně klienta

jako příliš velké zjednodušení, ale je potřeba si uvědomit, že úkolem tohoto programu je verifikovat průběh jednoho konkrétního spojení, kdy autentizace uživatele na jednom spojení neovlivní autentizaci na kterémkoliv dalším spojení. Díky tomu může být zavedeno zjednodušení, kdy uživatel může zadat buď platné nebo neplatné uživatelské jméno a platná nebo neplatná autentizační data.

Server může skrz druhý synchronní kanál posílat zjednodušené příkazy *UserAuthFailure* a *UserAuthSuccess*:

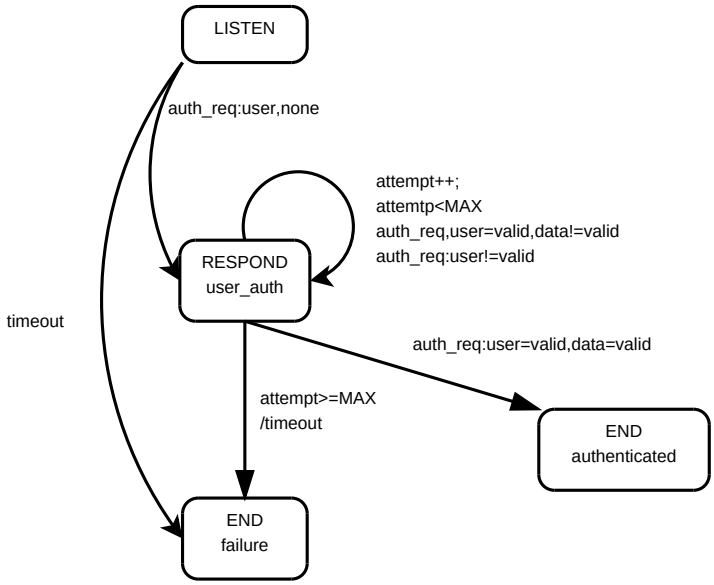
```
chan q1 = [0] of {bit, byte, bit};  /* user_name, auth_type, auth_data */
chan q2 = [0] of {bit, byte};       /* cmd_type, auth_meth */
```

Cílem verifikace této části protokolu bylo ověřit, že návrh neobsahuje žádné deadlocky, nevyvíjející se cykly a že oba procesy neskončí v nekorektních stavech. Na autentizaci nebyla kladena žádná další kritéria, která by šlo vyjádřit pomocí LTL logiky. Verifikace, jejíž výsledky lze nalézt v příloze na straně 129 prokázala, že tato část protokolu byla navržena korektně a neobsahuje žádné chyby.

Bezpečnostní rizika autentizace

Navržený autentizační mechanismus může být zranitelný vůči DoS a DDoS útoku, protože server čeká 30 sekund od ukončení TLS handshaku na první příkaz *UserAuthRequest* a stejnou dobu čeká od odeslání příkazu *UserAuthFailure* obsahující seznam podporovaných metod na další příkaz *UserAuthRequest*. Případný útočník se může pokusit otevřít velké množství spojení a snažit se vyčerpat systémové prostředky serveru či dosáhnout maximální povolené množství spojení na serveru.

Při návrhu protokolu se počítalo, že tento problém bude přenechán různým



Obrázek 4.14: Stavový model autentizace na serveru

packet filtrům jako je například iptables [4], které umožňují efektivně filtrovat počet pokusů o nové spojení podle různých kritérií. Některé packet filtry (včetně iptables) navíc umožňují spolupracovat se serverovými aplikacemi a provádět další penalizaci nových spojení pocházejících z IP adresy útočníka.

4.5.3 Dohadování vlastností spojení

Po ukončení autentizace může začít dohadování o novém datagramovém spojení. Pro dohadování nového spojení jsou použity systémové příkazy pro dohadování vlastností spojení, které jsou celkem čtyři: *Change_L*, *Confirm_L*, *Change_R* a *Confirm_R*, jejichž struktura je uvedena na obrázku 4.15. Inspirací pro tyto příkazy bylo dohadování vlastností spojení protokolu DCCP [25].

```
 0                   1         1         2     2             3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+---------------+-------------+---------------+---------------+--------
|    Op Code    |    Length   |    Feature    |     Value(s) ...
+---------------+-------------+---------------+---------------+--------
```

Obrázek 4.15: Struktura příkazů pro dohadování vlastností spojení

Za délkou příkazu, která nepřímo specifikuje kolik hodnot je v příkazu uvedeno, následuje identifikátor dohadované vlastnosti. Identifikátor vlastnosti je následován seznamem navrhovaných hodnot. Pořadí hodnot v příkazu

určuje jejich prioritu. Všechny čtyři příkazy se liší pouze v *Op Code*. Jejich význam je uveden v tabulce 4.2.

Op Code	Název	Význam
3	Change_L	Odesílatel navrhuje nastavení vlastnosti na své straně
4	Change_R	Odesílatel navrhuje nastavení vlastnosti na straně příjemce
5	Confirm_L	Příjemce potvrzuje odesílateli, že si může nastavit danou vlastnost
6	Confirm_R	Příjemce potvrzuje, že si nastavil danou vlastnost

Tabulka 4.2: Seznam příkazů pro dohadování vlastností spojení

Příkazy *Confirm_L* a *Confirm_R* smějí obsahovat buď jednu hodnotu, kterou potvrzují, nebo žádnou v případě, že si příjemce nevybral žádnou z navrhovaných variant. Pokud příjemce obdrží potvrzují příkaz obsahující seznam s více položkami, tak by měl ze seznamu použít první hodnotu a ostatní hodnoty ignorovat. Seznam vlastností spojení je uveden v tabulce 4.3.

Kód	Název	Velikost	Význam
0	Reserved		
1	FCID	uint8	Flow Control ID
2	CCID	uint8	Congestion Control ID
3	URL	string8	URL [6] datagramového spojení
4	Cookie	string16	Cookie
5	DED	string8	Data Exchange Definition
6	FPS	real32	FPS klienta

Tabulka 4.3: Seznam vlastností spojení

Dohadování vlastností spojení může být použito jak na TCP tak na UDP spojení, ovšem ne všechny vlastnosti má smysl dohadovat na TCP nebo naopak UDP spojení. Například je irelevantní snažit se dohadovat typ Congestion Control pro TCP spojení a naopak není možné dohadovat URL nového UDP spojení na již existujícím UDP spojení.

Dohadování o Flow Control a Congestion Control

Dohadování o Flow Control (FC) a Congestion Control (CC) může probíhat pouze na UDP spojení během jeho handshaku. Klient a server se musí

dohodnout jaké algoritmy budou používat a zároveň je požadováno, aby klient i server používaly stejný algoritmus pro FC a stejný algoritmus pro CC. Není možné, aby klient používal jiný algoritmus pro FC než jaký používá server. Pro dohadování FC a CC jsou definovány následující hodnoty:

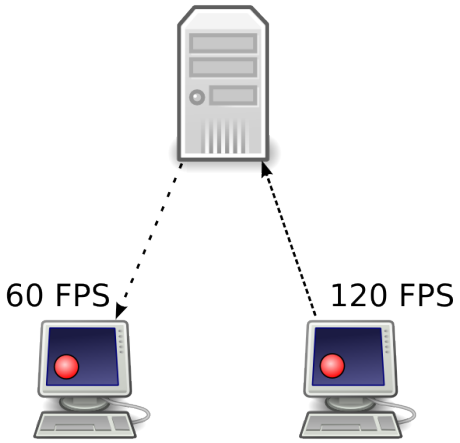
- **FC_RESERVED = 0** je rezervovaná hodnota, která by se nikdy neměla použita k nastavení FC
- **FC_NONE = 1** je hodnota, která říká, že klient i server nebudou používat FC
- **FC_TCP_LIKE = 2** je hodnota, která říká, že klient i server budou používat FC odvozený z TCP
- **CC_RESERVED = 0** je rezervovaná hodnota, která by se nikdy neměla použita k nastavení CC
- **CC_NONE = 1** je hodnota, která říká, že klient i server nebudou používat CC
- **CC_TCP_LIKE = 2** je hodnota, která říká, že klient i server budou používat CC odvozený z TCP

Dohadování o FPS

Klient by si měl se serverem průběžně dohadovat, jaký je momentálně schopný používat FPS pro zobrazování sdílených dat. Hlavní důvod použití dohadování je ve správném nastavení časovačů pro přeposílání paketů. Klientova hodnota FPS se může s časem měnit. Důvodem může být velké zatížení počítače, změna nastavení apod. Klient by měl dohadovat změnu FPS jenom v případě, že se změní více jak o 5 % a trvá déle jak 2/FPS s.

Další teoretické využití dohadování o FPS je ukázáno v příkladu, který je znázorněn na obrázku 4.16

V tomto příkladu bude první klient posílat pakety s frekvencí, která odpovídá 120 FPS, protože používá zobrazovací zařízení, které to umožňuje. Další klient je ovšem schopný zobrazovat informace maximálně s 60 FPS. Když by server přeposílal pakety obsahující polohu pohybujícího se objektu s frekvencí 120 Hz, tak by byl druhý klient schopný zobrazit pouze každou druhou polohu z obdržených paketů. Navíc síťový provoz generovaný na lince od serveru ke druhému klientovi je dvojnásobně velký než je potřeba a může na dané lince způsobit zahlcení přenosových cest. Když si druhý klient dohodne se serverem, že je momentálně schopný zobrazovat pouze 60 FPS, server by tomu měl přizpůsobit frekvenci odesílání paketů.



Obrázek 4.16: Dva Verse klienti s různým FPS

Dohadování o URL

Uniform Resource Locator (URL) [6] nového datagramového spojení si mohou klient a server dohodnout pouze na TCP spojení po autentizaci klienta. Z vlastního URL by měl klient a server využívat pouze schéma, adresu ve formě doménového jména nebo IP adresy a port. Ostatní položky (uživatelské jméno, dokument, atd.) by neměly být odesílány. Při přijetí příkazu obsahujícího URL s nadbytečnými položkami by měly být tyto položky ignorovány. Speciální význam má schéma URL, protože v něm se nastavují základní vlastnosti nového datagramového spojení. Schéma URL musí začínat řetězcem *verse*. Za ním následuje řetězec identifikující datagramový transportní protokol. Na konci schématu musí být uveden řetězec identifikující protokol používaný pro zabezpečení datagramového spojení. Všechny tři části schématu musí být odděleny znakem pomlčky. Podporovaný transportní protokol je v současné době pouze UDP, který je identifikován řetězcem *upd*. Pokud není vyžadováno zabezpečení datagramového spojení, pak je na konci schématu řetězec *none*. Při požadavku na zabezpečení datagramového spojení je možné použít protokol DTLS [31] [28], který je identifikován řetězcem *dtls*. Celé schéma je dle specifikace ukončeno řetězcem *://*. V současné verzi protokolu je tedy možné použít pouze dvě varianty schématu:

- `verse-udp-none://`
- `verse-udp-dtls://`

Číslo portu může být vyjádřeno buď jako číslo nebo jako textový identifikátor standardizovaný organizací IANA.

Dohadování o Cookie

Po dohodnutí nového UDP spojení začne server poslouchat na dohodnutém portu. Na tento port může kdokoliv odeslat paket a server by měl mít možnost ověřit, že daný paket pochází od klienta, který byl autentizován na TCP spojení. Stejně tak klient by měl mít možnost ověřit, že server je ten za koho se vydává. K tomuto účelu slouží náhodně vygenerovaná hodnota Cookie. Její doporučená délka je 16 znaků. Dohadovaná hodnota Cookie je přenášena jako string16 a může obsahovat i netisknutelné znaky. Příkazy *Change* a *Confirm* pro dohadování Cookie lze používat jak na TCP, tak na UDP spojení. Na TCP spojení slouží k dohadování nových hodnot Cookie a na UDP spojení slouží k prokázání identity klienta a serveru.

Dohadování o Data Exchange Definition

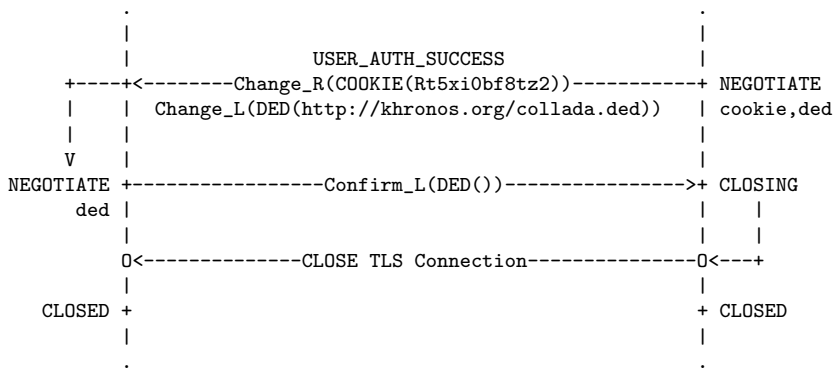
Data Exchange Definition (DED) je návrh mechanismu, který by měl umožňovat serveru kontrolovat konzistenci sdílených dat. Předchozí verze protokolu Verse kladla některá omezení na strukturu sdílených dat. U současného protokolu tomu tak není a datový model je navržen velmi obecně. Některým aplikacím tento přístup nemusí vyhovovat a mohou vyžadovat, aby se ostatní Verse klienti řídili dalšími pravidly při sdílení dat. Může být požadováno, aby sdílená data měla omezené rozsahy hodnot, počet položek, apod. K tomuto účelu je určený DED, což je odkaz na dokument, který by měl definovat tato pravidla.

Dohadování o datagramovém spojení

Vlastní dohadování o novém datagramovém spojení začne server tím, že do zprávy obsahující příkaz *UserAuthSuccess* přidá i příkaz *Change_L* obsahující jeden návrh Cookie a příkaz *Change_R* obsahující jeden návrh DED. Server zasláním těchto dvou zpráv říká klientovi, jakou bude vyžadovat Cookie při poslání prvního prvního paketu na datagramovém spojení a jaká jsou další pravidla při sdílení dat na serveru.

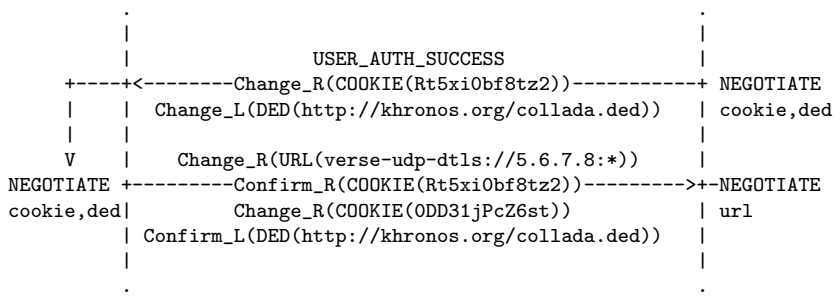
Když není klient schopen dodržovat pravidla popsaná v DED, tak by měl odpovědět příkazem uvedeným na obrázku 4.17 a server by měl s klientem ukončit spojení.

Pokud je klient schopný dodržovat pravidla definovaná v DED, tak by měl v následující odpovědi serveru zaslat příkaz *Confirm_R* obsahující navrhované DED a příkaz *Confirm_L* potvrzující navrhované Cookie. Dále tato



Obrázek 4.17: Příklad odpovědi klienta na nepodporované DED

zpráva musí obsahovat příkaz s návrhem URL, které musí obsahovat IP adresu serveru pomocí které je klient k tomuto serveru připojený. Místo čísla portu by měla být v URL uvedena pouze hvězdička, protože číslo portu vybere následně server. Nakonec zpráva může obsahovat příkaz *Change_R* s návrhem Cookie pokud bude klient vyžadovat aby i server prokázal na datagramovém spojení svoji identitu. Tato část dohadování o novém spojení je popsána na obrázku 4.18.



Obrázek 4.18: Příklad odpovědi klienta na podporované DED

Když server přijme zprávu s potvrzením DED a Cookie, tak náhodně vybere volný UDP port a začne na něm poslouchat. Server nemusí vyslyšet požadavek klienta na vytvoření zabezpečeného nebo nezabezpečeného datagramového spojení a může se řídit vlastní bezpečnostní politikou. Jediné, co je pro server z návrhu URL směrodatné, je navrhovaná IP adresa, protože tu server doplní o platné schéma a použitý port a pošle klientovi v novém návrhu. Server totiž nesmí potvrdit klientův návrh na URL už jen proto, že klient měl ve svém návrhu místo portu uveden znak hvězdičky.

Server tedy odešle klientovi zprávu s příkazem *Confirm_R(URL())*, který

klientovi signalizuje, že neakceptuje jeho návrh URL obsahující místo portu znak hvězdičky. Server do paketu přidá příkaz *Change_L* s vlastním návrhem platného URL a pokud to klient vyžadoval, tak i potvrzení Cookie. Na tuto odpověď má server limit 30 sekund. Při jeho nedodržení by měl klient spojení se serverem ukončit.

V případě, že se serveru nepodaří otevřít volný UDP port, tak odešle klientovi zprávu s prázdným příkazem *Confirm_R(URL())* a ukončí spojení.

Klient se po přijmutí zprávy obsahující návrh platného URL pokusí provést handshake na UDP spojení a teprve když uspěje, tak serveru navrhované URL potvrdí. Kdyby handshake na UDP spojení z nějakého důvodu selhal, tak klient odpoví serveru na jeho návrh URL zprávou obsahující prázdný příkaz *Confirm_L(URL())*. Server a klient by měly následně uzavřít spojení. Handshake na UDP spojení bude podrobně popsán v kapitole 4.6. Celý průběh navázání spojení mezi klientem a serverem je popsán na obrázku 4.21.

Server by měl čekat na potvrzení URL výjimečně 60 sekund, protože handshake na UDP spojení má také několik fází a každá má svůj 30 sekundový časový limit.

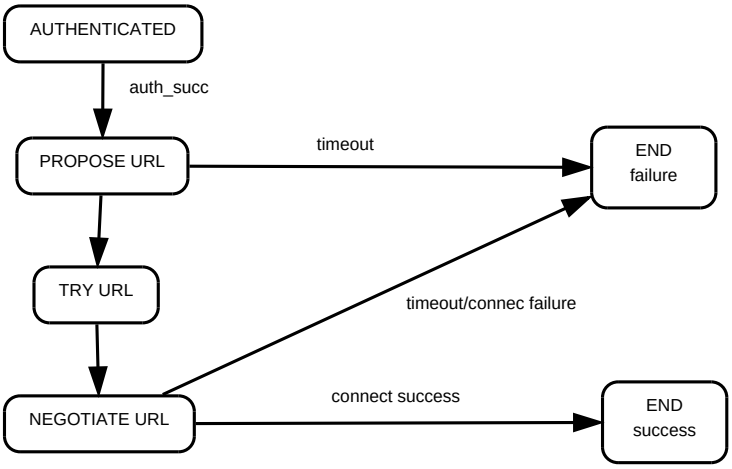
Delegace datagramového spojení

Důvodem použití URL pro dohadování nového datagramového spojení je i teoretická možnost delegovat datagramové spojení na jiný server (SlaveServer) a rozložit zátěž hlavního serveru (MasterServer). Další využití tohoto přístupu by mohlo být přepojování klientů na SlaveServery podle geolokace, uživatelského jména, dohodnutého DED, apod. Touto problematikou se zabývají práce [26] [8]. Delegování datagramového spojení by komplikovalo dohadování o novém spojení. Předně by bylo nutné vytvořit zabezpečený komunikační kanál mezi MasterServerem a SlaveServerem pomocí kterého by se přenášely informace o autentizovaných klientech, Cookie, apod. SlaveServer by se navíc musel v některých případech chovat vůči MasterServeru jako Verse klient, což by dále komplikovalo návrh a implementaci tohoto protokolu.

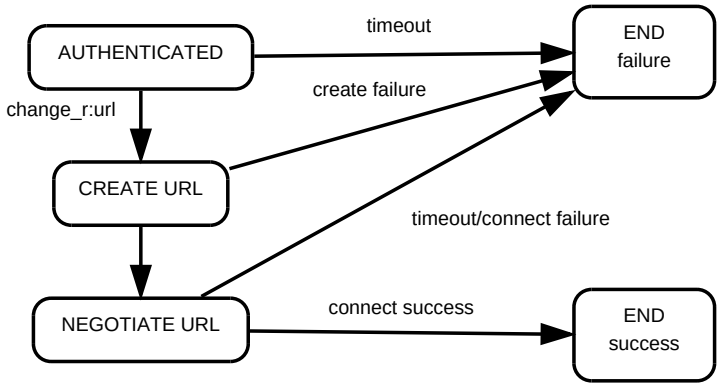
Verifikace dohadování datagramového spojení

Část handshaku dohadující nové datagramové spojení byla verifikována v samostatném modelu. Jelikož je primárním účelem této části handshaku dohodnout nové datagramové spojení, tak byl proces dohadování spojení zjednodušen pouze na dohadování o URL. Cookie a DED jsou pouze další vlastnosti, které zlepšují zabezpečení a funkční vlastnosti protokolu. Před vytvořením modelu byly vytvořeny stavové modely klienta a serveru, které

jsou uvedeny na obrázcích 4.19 a 4.20



Obrázek 4.19: Stavový model dohadování datagramového spojení na straně klienta



Obrázek 4.20: Stavový model dohadování datagramového spojení na serveru

Z těchto dvou stavových modelů byl vytvořen verifikační model, který je uveden v příloze na straně 131. Díky výše uvedeným zjednodušením bylo možné zprávy mezi procesem klienta a serveru přenášet pomocí dvou synchronních kanálů:

```
mtype = {none, change_l, change_r, confirm_l, confirm_r}
/* cmd_type, url, cmd_type, url */
chan q1 = [0] of {mtype, byte, mtype, byte};
chan q2 = [0] of {mtype, byte, mtype, byte};
```

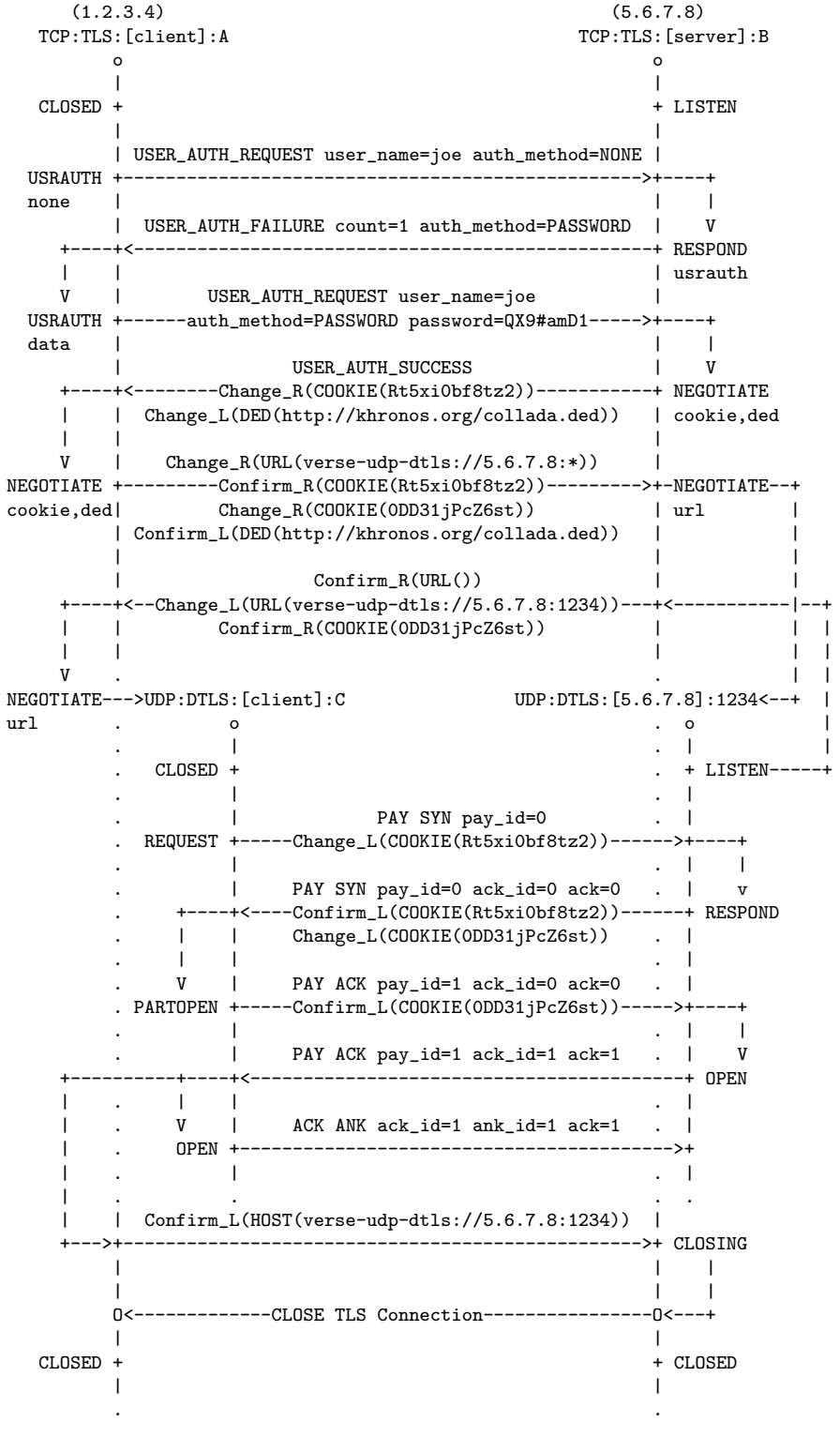
Přenášená zpráva mohla obsahovat buď jeden nebo dva příkazy *Change_L*, *Change_R*, *Confirm_L*, *Confirm_R*, které mohly navrhovat nebo potvrzovat

URL vyjádřený jedním bytem. URL mohlo nabývat tří hodnot. Hodnota *NONE_URL* byla použita v kombinaci s příkazy *Confirm* k zamítání navrhovaných hodnot. *VALID_URL* vyjadřovala platné URL a *UNVALID_URL* byla použita jako návrh klienta na URL nového datagramového spojení.

Výsledky verifikace uvedené v příloze na straně 135 prokázaly, že se v této části protokolu nenachází žádný deadlock ani nevyvíjející se cykly.

Bezpečnostní rizika dohadování datagramového spojení

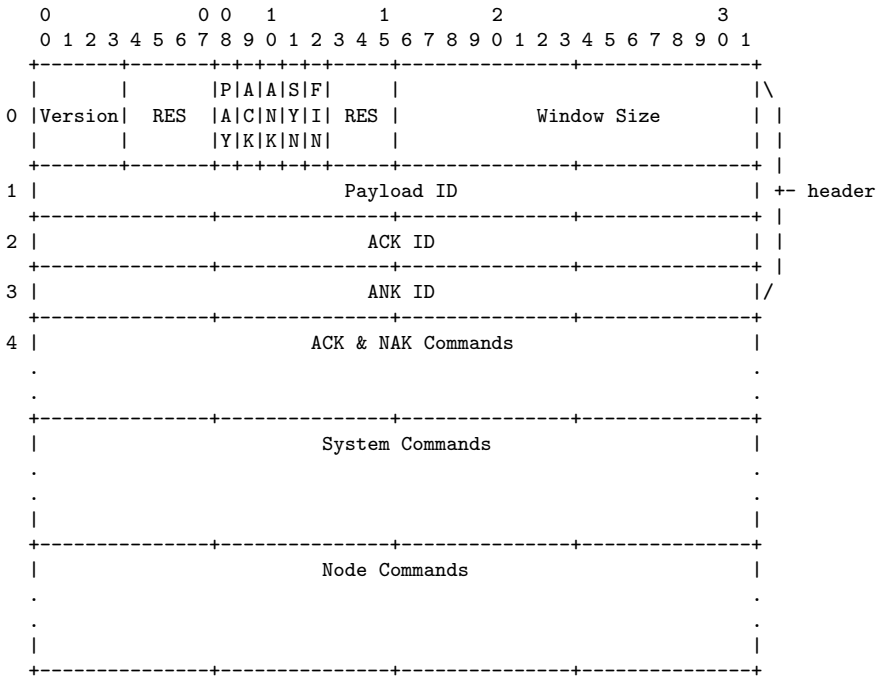
Při vlastním dohadování nového spojení by neměla hrozit nějaká reálná bezpečnostní rizika. Problém může ovšem nastat v případě, že server umožňuje použít nezabezpečené datagramové spojení i na linkách, které nejsou zabezpečeny jiným způsobem (IPsec, localhost, apod.). Útočník může na nezabezpečené datagramové spojení zaútočit dvěma způsoby. Může se pokusit uhádnout číslo portů nově otevřeného spojení a toto spojení ukrást. Tento způsob by měl být předem odsouzen k nezdaru, protože klient by musel uhodnout s číslem portu současně i hodnotu Cookie a taková možnost je krajně nepravděpodobná. Větším rizikem je útok typu Man-in-the-middle, protože útočník může zachytit paket obsahující číslo cílového portu a Cookie. Útočník může následně tyto informace použít k připojení se k serveru. Z tohoto důvodu se nedoporučuje používat nezabezpečené datagramové spojení na síti Internet.



Obrázek 4.21: Příklad kompletního handshaku

4.6 Datagramové spojení

Protože v se v současném návrhu protokolu počítá na transportní vrstvě pouze s protokolem UDP, tak se bude nadále uvažovat na datagramovém spojení pouze protokol UDP. Veškeré příkazy přenášené na UDP spojení je potřeba přenášet v paketech, které mají strukturu uvedenou na obrázku 4.22.



Obrázek 4.22: Struktura Verse paketu

Paket se skládá z hlavičky za níž musí následovat potvrzovací příkazy. Za potvrzovacími příkazy následují systémové příkazy a teprve po nich tzv. uzlové příkazy přenášející užitečná data. Mezi potvrzovacími příkazy, systémovými příkazy a ani uzlovými příkazy nedochází k žádnému vyplňování. Na obrázku 4.22 je zarovnání na délku 4 bytů pouze pro lepší přehlednost.

Hlavička paketu začíná verzí protokolu *Version*. Současná verze protokolu má *Version = 1*. Následují 4 bity, které slouží buď jako rezerva pro příliš velká čísla verze protokolu nebo případně pro další příznaky. Za rezervou následují příznaky:

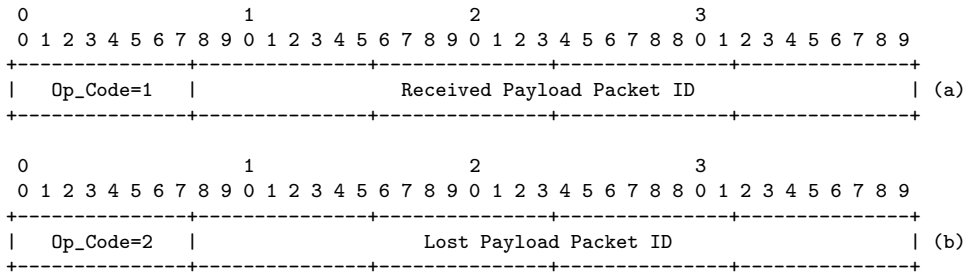
- *PAY* značí, že v paketu jsou přenášena užitečná data (payload data) a položka *Payload ID* obsahuje platnou hodnotu. V opačném případě by měla být vyplněna nulami.

- *ACK* udává, že paket obsahuje speciální *ACK* nebo *NAK* příkazy a položka *ACK ID* obsahuje platnou hodnotu. V opačném případě by měla být vyplněna nulami.
- *ANK* je nastavený, když položka *ANK ID* obsahuje platnou hodnotu. V opačném případě by měla být vyplněna nulami.
- *SYN* se používá během handshaku
- *FIN* se používá během ukončení spojení.

Když si klient a server během handshaku na UDP spojení dohodnou nějaký algoritmus pro Flow Control, tak se pro signalizaci velikosti okénka používá položka *Windows Size*. Přenášená hodnota může udávat velikost v bytech nebo jejich násobcích podle dohodnutého algoritmu.

Payload ID se používá jako jedinečný identifikátor payload paketů. S každým odeslaným payload paketem se inkrementuje příslušný čítač. Když jeho hodnota dosáhne maxima: $2^{32} - 1$, tak se hodnota čítače nastaví na nulu. Stejné pravidlo platí i pro *ACK ID*, které funguje jako jedinečný identifikátor potvrzovacích paketů. Položka *ANK ID* obsahuje identifikátor naposledy potvrzeného payload paketu.

Pokud příjemce potřebuje potvrdit přijetí nebo ztrátu payload paketu, tak za hlavičkou vloží potvrzovací příkazy, které se od ostatních příkazů liší v tom, že se za jejich *Op_Code* nenachází délka příkazu. Potvrzovací příkazy mají konstantní délku a k jejich případné komprimaci dochází jiným způsobem než u ostatních příkazů. Struktura potvrzovacích příkazů je uvedena na obrázku 4.23. Jejich funkce a použití budou podrobněji popsány v následujících částech.



Obrázek 4.23: Struktura příkazu ACK (a) a NAK (b)

4.7 Handshake datagramového spojení

Než si mohou klient a server vyměnit první data, tak mezi nimi musí proběhnout čtyřcestný handshake, jehož návrh byl inspirován handshakem protokolu DCCP. Handshake na datagramovém spojení umožňuje klientovi i serveru prověřit průchodnost přenosových cest před tím než si alokují nezbytné systémové prostředky pro zajištění částečné spolehlivosti datagramového spojení. Během handshaku si také mohou dohodnout další vlastnosti spojení jako jsou použité algoritmy pro FC a CC.

Klient může začít handshake až po té, co od serveru obdrží na TCP spojení návrh URL datagramového spojení. Vlastnímu handshaku musí předcházet DTLS handshake, pokud je schéma serverem navrženého URL zakončeno řetězcem *dtls://*. DTLS protokol i jeho handshake jsou detailně popsány v [31]. DTLS se příliš neliší od TLS, které bylo popsáno v části TLS handshake. Hlavní rozdíly mezi TLS a DTLS jsou popsány v článku [28].

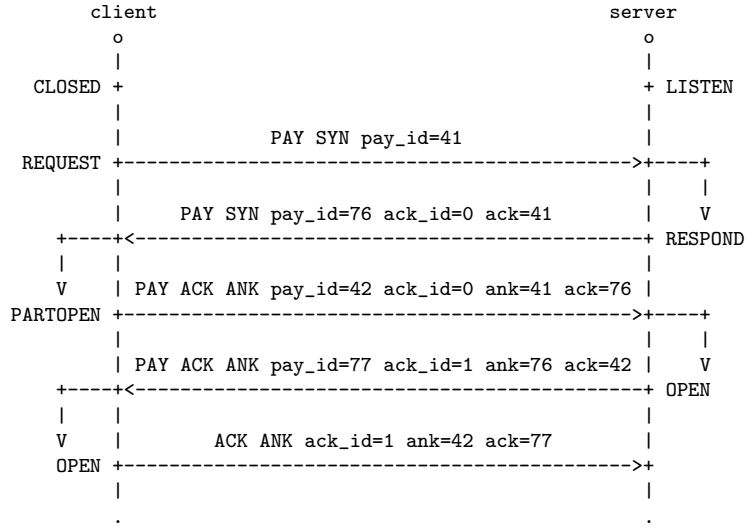
Součástí DTLS handshaku je i objevování PMTU, které umožňuje zjistit maximální MTU na obou linkách mezi klientem a serverem. V případě, že je použita nezabezpečená varianta datagramového spojení, tak je vhodné, aby klient i server použili vlastní PMTU. Na moderních operačních systémech k tomu není potřeba mít oprávnění administrátora, ale je možné zjistit PMTU pomocí příslušných systémových volání nad sokety daného spojení. Získání PMTU umožňuje posílání velkých paketů bez rizika fragmentace paketů na dané lince. PMTU zabezpečeného spojení je vždy menší než PMTU nezabezpečeného spojení na stejné lince, protože DTLS do všech paketů přidává vlastní hlavičku. DTLS přidává do paketu kromě hlavičky i MAC a tím dále zmenšuje prostor pro přenášená data, což DTLS umožňuje kompenzovat pomocí komprese přenášených dat.

Příklad výměny paketů během handshaku je uveden na obrázku 4.24. V tomto příkladě nejsou pro přehlednost uvedeny žádné další systémové příkazy pro dohadování vlastnosti spojení (Cookie, Flow Control, Congestion Control).

4.7.1 První část handshaku

Během handshaku se klient i server nacházejí v několika stavech. Spojení zahajuje klient ve stavu *REQUEST* zasláním paketu, který má nastaven příznaky *PAY* a *SYN*. Jedná se tudíž o prázdný payload paket, který má počáteční *Payload ID* nastaveno na náhodnou hodnotu. Příznak *SYN* říká, že se jedná o paket zahajující spojení.

Pokud první paket přijatý serverem obsahuje podporovanou verzi pro-



Obrázek 4.24: Příklad handshaku na datagramovém spojení

tokolu a příznaky *PAY* a *SYN*, tak by se měl server přepnout ze stavu *LISTEN* do stavu *RESPOND* a odpovědět odesláním paketu obsahujícího příznaky *PAY*, *SYN* a *ACK*. Jedná se opět o payload paket s náhodně zvoleným *Payload ID*, který k sobě přibaluje i potvrzovací paket s příkazem *ACK* potvrzující přijetí prvního paketu. Počáteční hodnota *ACK ID* přibaleného potvrzovacího paketu by měla být nulová.

Jelikož je na transportní vrstvě použit nespolehlivý datagramový protokol UDP, tak je nezbytné, aby klient přeposlal ztracený paket v nezměněné podobě. To znamená, že nesmí být inkrementována ani jinak změněna žádná položka v hlavičce a dalších příkazech. Ztráta paketu je detekována pomocí vypršení časovače. Počáteční hodnota časovače je jedna sekunda. Při každé další ztrátě paketu (vyjádřeno proměnnou *step*) se hodnota časovače zvyšuje podle následujícího algoritmu:

begin
  $init\_timout := 1;$
  $back\_off := 2;$
  for $i := 0$ **to** *step* **step** 1 **do**
    $back\_off := back\_off * 2;$
  end
  if $back\_off < MAX\_BACK\_OFF$
    then
      $back\_off := back\_off - 1;$

else

$back\_off := MAX\_BACK\_OFF;$

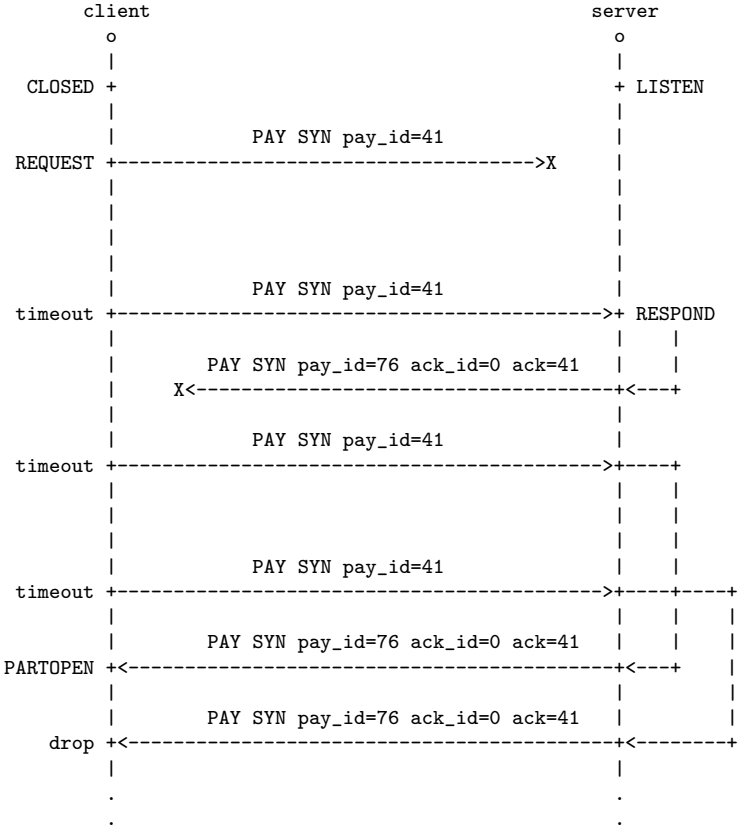
fi

$timout := init\_timeout + back\_off * (rand()/(MAX\_RAND + 1));$

end

kde *rand()* je funkce vracející náhodnou hodnotu nebo pseudonáhodnou hodnotu s náhodnou počáteční hodnotou v rozsahu $\langle 0, MAX\_RAND \rangle$.

V první části handshaku může dojít při výměně paketů ke čtyřem možným scénářům, které jsou uvedeny na obrázku 4.25.



Obrázek 4.25: Příklad 4 možných scénářů během první části handshaku

Z obrázku je patrné, že k dosažení spolehlivosti handshaku je nutné, aby server po přepnutí do stavu *RESPOND* stále odpovídal i na pakety, které odeslal klient ve stavu *REQUEST*. Klient nesmí během stavu *REQUEST* poslat více jak 10 paketů a v tomto stavu se nesmí nacházet déle jak 30 sekund. Pokud do této doby neobdrží správnou odpověď od serveru, tak by měl klient po TCP spojení poslat serveru prázdný příkaz *Change_L(URL)*

signalizující, že se spojení se serverem na UDP nezdařilo a následně ukončit se serverem komunikaci.

Podobně by měl server odpovědět maximálně na 10 paketů, které mu poslal klient ve stavu *REQUEST*. Neobdrží-li server paket odeslaný klientem ze stavu *PARTOPEN* do 30 sekund od doby, kdy se přepnul do stavu *RESPOND*, tak by měl s klientem na UDP spojení ukončit komunikaci. Server by měl zároveň odpovídat pouze na pakety, které obsahují příkaz *Change_L(Cookie)* s dohodnutou hodnotu Cookie. Server by měl odpovídat i na pakety, které pocházejí z jiné IP adresy než na které vede s klientem komunikaci po TCP, protože klient může mít více síťových rozhraní a přestože je to velmi nepravděpodobné, tak TCP spojení může být směrováno přes jiné síťové rozhraní klienta než UDP spojení. Zároveň server musí ve stavu *RESPOND* odpovídat na dotazy z jedné kombinace IP adresa port stále stejným paketem.

4.7.2 Druhá část handshaku

Po té, co klient obdrží od serveru paket s platnou odpovědí, by se měl přepnout do stavu *PARTOPEN*. Paketem s platnou odpovědí se rozumí paket obsahující příznaky *PAY*, *SYN* a *ACK* a systémový příkaz *Ack* potvrzující přijetí prvního paketu. V tomto paketu by měl server poslat i příkaz *Confirm_L(Cookie)* s potvrzením platnosti Cookie a případně i svoji Cookie v příkazu *Change_L(Cookie)*, pokud to bylo dohodnuto na TCP spojení. Za předpokladu, že jedna z Cookie neodpovídala tomu, co bylo dohodnuto na TCP spojení, by měl klient přijatý paket ignorovat.

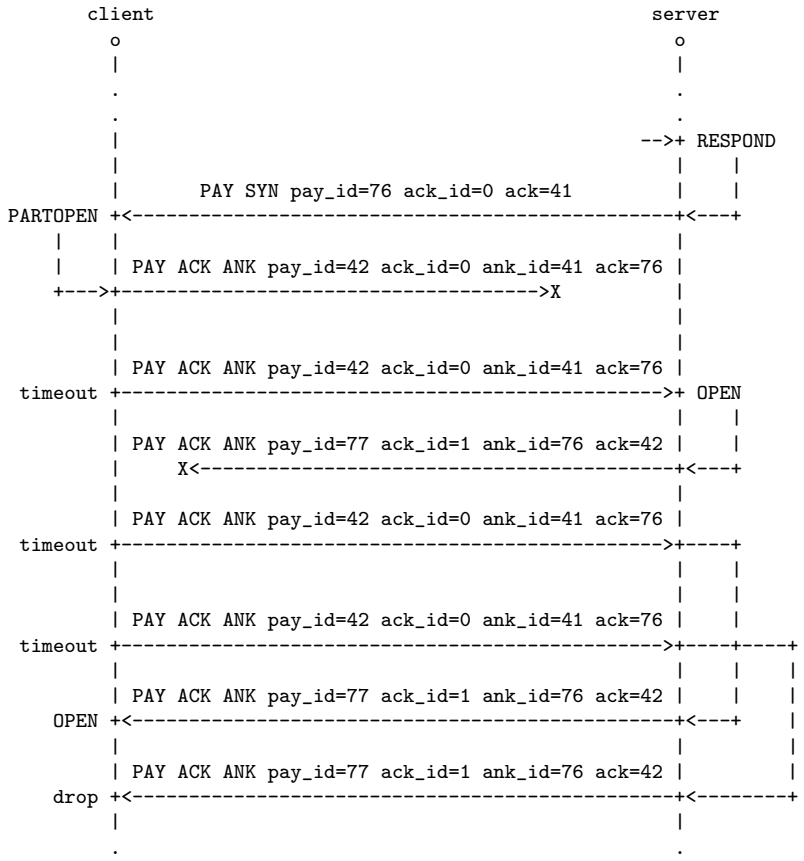
V režimu *PARTOPEN* by měl klient posílat pakety s příznaky *PAY*, *ACK* a *ANK*, kdy hodnota *Payload ID* je o jedna větší než byla u paketů zasílaných ve stavu *REQUEST*. Dále by měl paket obsahovat příkaz *ACK* potvrzující přijetí payload paketu a platnou položku *ANK ID* signalizující serveru, že už není nadále nutné potvrzovat přijetí prvního paketu. Paket by měl obsahovat i příkaz *Confirm_L(Cookie)* v případě, že se server prokázal platnou Cookie.

Klient může po přepnutí do režimu *PARTOPEN* obdržet odpovědi na pakety odeslané ještě ve stavu *REQUEST*. Takové pakety by měl klient ignorovat.

V druhé části handshaku by se měl server přepnout do stavu *OPEN*, pokud přijme paket pocházejí ze stejné kombinace IP adresa:port jako byly pakety přijaté ve stavech *LISTEN* a *RESPOND*. Zároveň musí přijatý paket obsahovat potvrzení payload paketu, který server poslal klientovi v první části handshaku. Server by měl odpovědět payload paketem, který bude zároveň potvrzovat přijatý payload paket. Server by měl alokovat systémové prostředky potřebné pro resend mechanismus teprve ve stavu *OPEN*, protože

tehdy má server jistotu, že komunikace s daným klientem je možná. Klient se přepne do stavu *OPEN* po obdržení výše zmíněného potvrzovacího paketu.

Všechny možné scénáře výměny paketů během druhé části handshaku jsou uvedeny na obrázku 4.26.



Obrázek 4.26: Příklad 4 možných scénářů během druhé části handshaku

Z obrázku je patrné, že klient opět může obdržet duplikované pakety po přepnutí do stavu *OPEN*. Takové pakety by měl klient ignorovat.

4.7.3 Dohadování na datagramovém spojení

Vlastnosti datagramového spojení (Congestion Control, Flow Control, apod.) není možné dohadovat na původním TCP spojení, protože by to velmi komplikovalo delegaci datagramového spojení na jiný server, jak to bylo popsáno v kapitole 4.5.3. Každý server by měl mít možnost si s klientem dohadovat vlastnosti spojení podle vlastního nastavení.

Dohadování vlastností na datagramovém spojení se liší od dohadování na TCP spojení v tom, že není zaručena spolehlivost doručení dat. Z toho důvodu je nutné zajistit přeposílání dohadovacích příkazů. Během handshaku detekuje ztrátu paketu pouze klient na základě vypršení časovače a klient ztracený paket přeposílá v nezměněné podobě. Ve stavu *OPEN* se používá jiný mechanismus detekce ztráty paketu.

Aby se odesílatel s příjemcem dohodli na vlastnostech pomocí dohadovacích příkazů, tak musí dodržovat několik pravidel. Příjemce musí na dohadovací příkazy odpovědět ihned. Když odesílatel detekuje ztrátu paketu obsahujícího dohadovací příkazy, tak by měl přeposlat pouze příkazy *Change_L* a *Change_R*. Navíc by měl odesílatel zopakovat odeslání dohadovacích příkazů *Change_L*, *Change_R*, když na ně neobdrží odpověď do doby, která je rovna dvojnásobku *SRTT*.

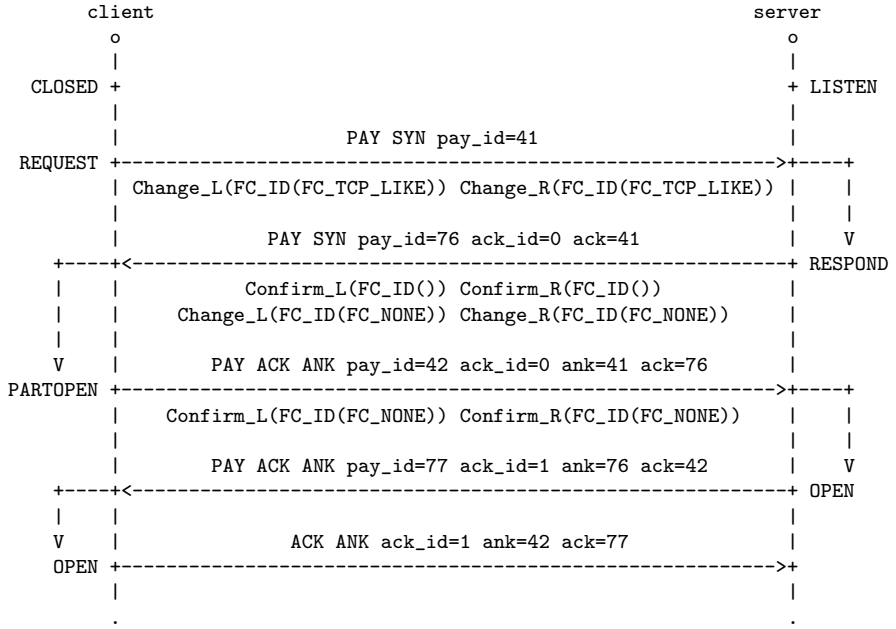
4.7.4 Dohadování o Flow Control

Během handshaku si obě strany dohodnou jaký budou používat algoritmus pro Flow Control (FC) neboli kontrolu zahlcení příjemce. Zahlcením je více ohrožený server, takže klient sice může navrhnout použitý algoritmus, ale určující slovo při nastavení používaného algoritmu by měl mít server. Pro správné fungování FC je nezbytné, aby obě strany používaly totožné algoritmy. Na obrázku 4.27 je uveden příklad dohadování.

Na obrázku 4.27 je vidět, že klient může postal návrhy na FC, ale ty jsou v tomto případě serverem zamítnuty. Server následně posílá své návrhy na algoritmu FC, které musí klient potvrdit. Server může navrhnout použití více algoritmů, ale oba příkazy *Change_L(FC_ID)* i *Change_R(FC_ID)* by měly obsazovat totožný seznam navrhovaných hodnot. Vždy by mělo dojít k navržení a potvrzení FC pro příjemce i odesílatele, aby nevznikly nejasnosti, jaký algoritmus bude daná strana používat.

Pokud klient navrhované algoritmy nepotvrdí (pošle serveru prázdné příkazy *Change_L(FC_ID)* a *Change_R(FC_ID)*), protože je například neimplementuje, tak by se server neměl přepnout do stavu *OPEN*, ale měl by se přepnout do stavu *CLOSEREQ* a spojení s klientem přátelsky ukončit. Příklad přátelského nedohodnutí je uveden na obrázku 4.28.

Pokud se klient se serverem nedohodnou na jiné vlastnosti spojení během handshaku, která je nezbytná pro správné fungování datagramového spojení, je potřeba spojení ukončit stejným způsobem jako v případě nedohodnutí se na Flow Control. Ukončení spojení bude podrobněji vysvětleno v části 4.9.



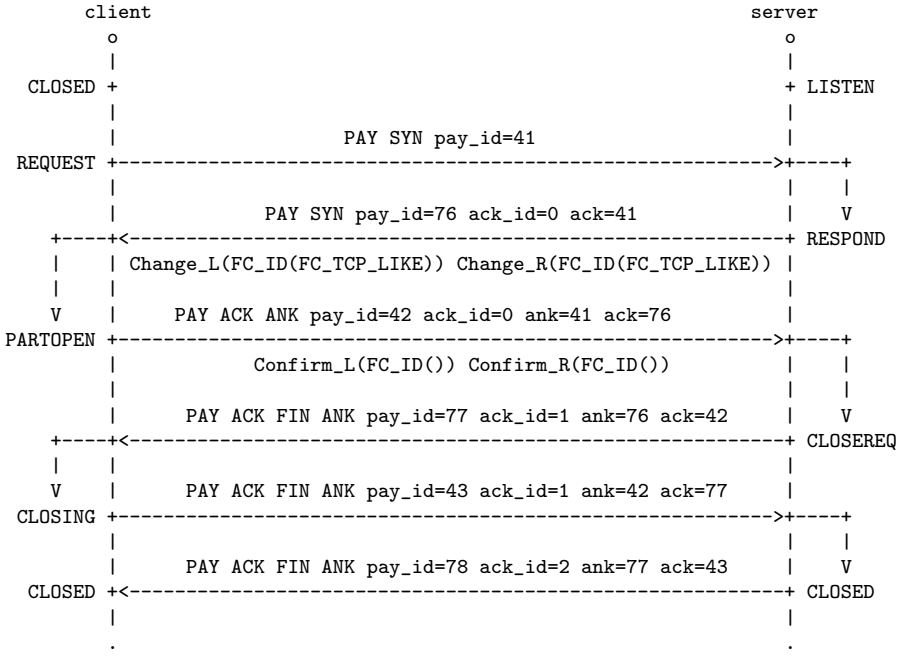
Obrázek 4.27: Příklad dohadování o Flow Control

4.7.5 Dohadování o Congestion Control

Congestion Control (CC) by měl být určován především klientem, protože se předpokládá, že autor aplikace dokáže určit, jaký algoritmus CC bude nejlépe vyhovovat povaze přenášených dat. Konečné slovo při určování vlastnosti spojení má ovšem zase server. Příklad dohadování o algoritmu pro CC je uveden na obrázku 4.29.

Dohadování o CC začne klient zasláním příkazu *Change_L(CC_ID)* se seznamem navrhovaných algoritmů pro linku klient-server. Server by měl ze seznamu vybrat nějaký algoritmus a potvrdit jeho použití zasláním příkazu *Confirm_L(CC_ID)*. Zároveň by měl server stejný algoritmus navrhnout pro linku server-klient. Klient by měl tuto volbu potvrdit. Když se klient a server nedohodnou na použitém algoritmu pro CC, tak by měl server ukončit komunikaci s klientem stejným způsobem jako je ukončeno spojení po neúspěšném dohadování o Flow Control.

Dohadování o Flow a Congestion Control je možné provést pouze během handshaku. Ve stavu *OPEN* již není možné používané algoritmy měnit. Flow ani Congestion Control není možné nastavovat na TCP spojení před zahájením vlastního datagramového handshaku, protože se počítá s budoucí možností delegovat datagramové spojení na jiný server a každý server by měl mít možnost si zvolit vlastní algoritmy.



Obrázek 4.28: Příklad nedohodnutí algoritmu Flow Control

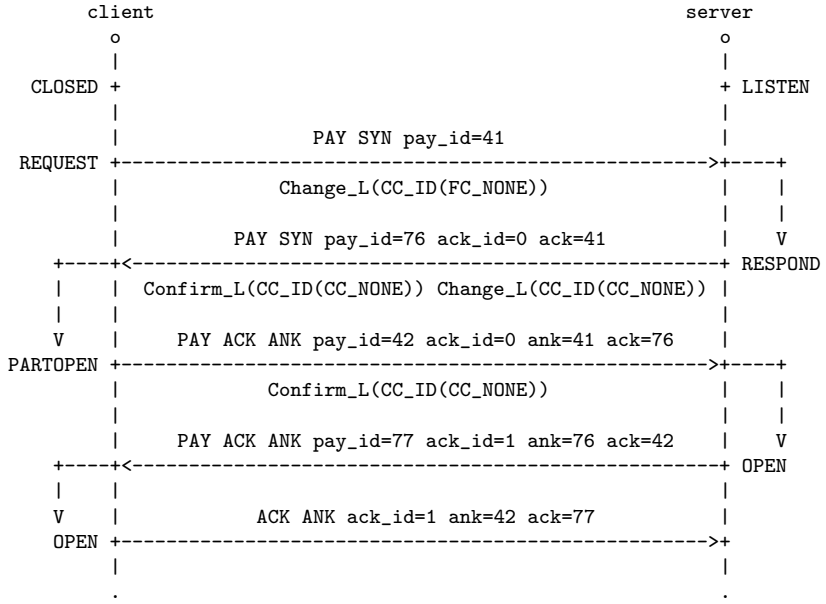
4.7.6 Bezpečnostní rizika handshaku

Na handshake datagramového spojení se může útočník pokusit provést DoS/DDoS útok. Případný útok je ztížený skutečností, že server otvírá náhodný UDP port. Přístup k tomuto portu může samotný server omezit na konkrétní IP adresu, přestože bylo ukázáno, že to může některým klientům znemožnit navázat UDP spojení. Jako obrana před DoS/DDoS útoky se doporučuje použít nějaký paket filtr obdobným způsobem jako je tomu u TCP spojení.

Samotný handshake poskytuje poměrně dobrou obranu proti ukradení datagramového spojení, pokud je datagramové spojení zabezpečené pomocí DTLS protokolu. V opačném případě je klient vystaven riziku útoku Man-in-the-middle. Z tohoto důvodu se nedoporučuje používat nezabezpečenou variantu datagramového spojení na na linkách, které nejsou zabezpečené nějakým jiným způsobem.

4.8 Resend mechanismus

Jelikož se na transportní vrstvě používá nespolehlivý datagramový protokol UDP a pro real-timové sdílení dat je vyžadována částečná spolehlivost



Obrázek 4.29: Příklad dohadování o Congestion Control

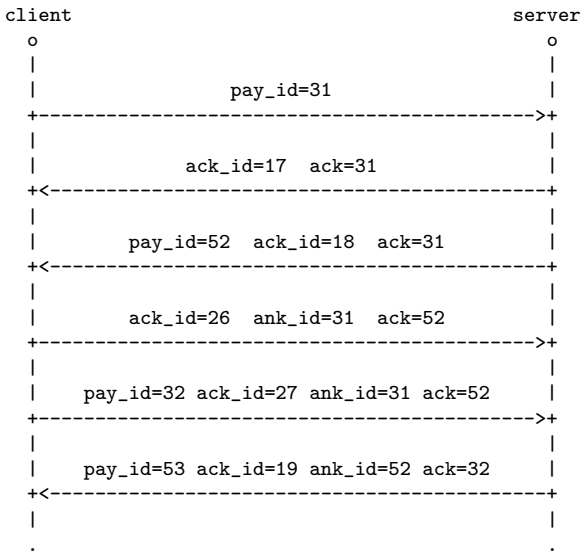
přenášených dat, je potřeba implementovat vlastní resend mechanismus na aplikační vrstvě. Resend mechanismu přejímá některé myšlenky z původního protokolu Verse, ale celkově byl od základu přepracován s důrazem na spolehlivost a robustnost.

Při návrhu resend mechanismu byl použit stejný princip, který byl popsán na straně 25. Data jsou v novém protokolu přeposílána také pouze v případě, že jsou aktuální, ale detekce ztráty dat a její signalizace jsou navrženy odlišným způsobem.

4.8.1 Pozitivní potvrzování

Na obrázku 4.30 je zjednodušený příklad komunikace mezi klientem a serverem ve stavu *OPEN*, kdy nedošlo k žádné ztrátě paketů a klient i server si potvrzují pouze přijetí payload paketů pomocí příkazů *ACK*. Na tomto příkladu budou popsány a vysvětleny základní principy a mechanismy pozitivního potvrzování paketů pomocí příkazu *ACK*.

Klient si po odeslání payload paketu (obsahuje příznak PAY) s $ID = 31$ uloží jeho obsah do historie odeslaných paketů, protože při jeho případné ztrátě musí být schopný přeposlat aktuální data ze ztraceného paketu. Server po přijetí payload paketu s $ID = 31$ okamžitě odešle klientovi potvrzovací paket (obsahuje příznak ACK) s $ACK\_ID = 17$. Příjemce by měl potvrzo-



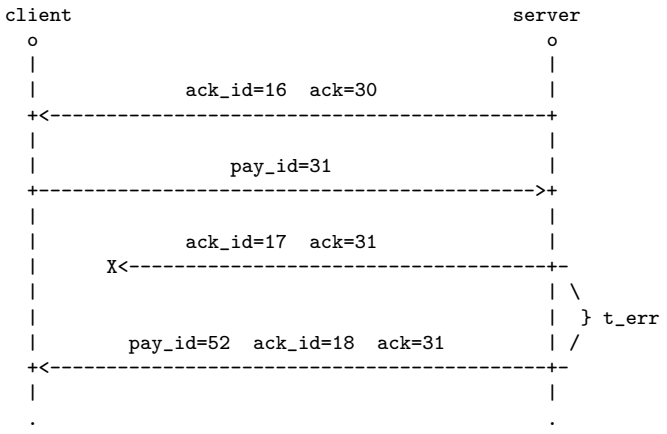
Obrázek 4.30: Příklad pozitivního potvrzení

vací paket odeslat ihned, aby mohl odesílatel spočítat korektní hodnotu RTT. Po nějaké době pošle server klientovi data v payload paketu s $ID = 52$. Do tohoto paketu je přibalen i obsah předchozího potvrzovacího paketu (obsahuje příznaky *PAY* i *ACK*), protože je potřeba maximalizovat pravděpodobnost doručení potvrzovacích příkazů. Toto přibalování paketů se nazývá *Piggybacking*.

Když klient přijme potvrzení o doručení payload paketu s $ID = 31$, obsah tohoto paketu odstraní z historie odeslaných paketů. Klient na přijetí payload paketu s $ID = 52$ zareaguje taktéž odesláním potvrzovacího paketu. Jeho hlavička bude navíc obsahovat platnou položku s $ANK\_ID = 31$ (příznak *ANK* je nastaven), kterou příjemce (v tomto případě klient) oznamuje odesílateli, že největší potvrzené *Payload_ID* má hodnotu *31* a již není nadále nutné potvrzovat přijetí nebo ztrátu tohoto a předchozích payload paketů.

Příjemce při přijetí payload paketu nemusí posílat potvrzovací paket v samostatném paketu. V případě, že má sám v daný okamžik k odeslání nějaká data, tak potvrzovací paket může být rovnou přibalen k payload paketu.

Příjemce by neměl spočítat RTT při přijetí každého potvrzovacího paketu ale pouze v případě, že nedošlo ke ztrátě předchozího potvrzovacího paketu, jak je uvedeno na obrázku 4.31. Kdyby příjemce spočítal RTT při přijetí potvrzovacího paketu *ACK ID = 18*, tak by se při výpočtu dopustil chyby t_{err}. Příjemce detekuje ztrátu potvrzovacího paketu, když potvrzovací paket nemá očekávané *ACK ID*.



Obrázek 4.31: Příklad ztráty paketu s pozitivním potvrzením

4.8.2 Negativní potvrzování

Detekce ztráty paketů se v novém protokolu Verse provádí dvěma způsoby. První způsob vychází z původního protokolu Verse 3.3, kdy ztráta paketu je detekována příjemcem. Druhý způsob používá pro detekci ztráty paketu Retransmit Timeout (RTO) interval na straně odesílatele paketu. K výpočtu se používá poměrně konzervativní metoda popsaná v [22] zamezující zbytečnému přeposílání dat. *RTO* se vypočítá ze Smoothed Round Trip Time (SRTT):

$$SRTT \leftarrow \begin{cases} RTT, & SRTT = 0 \\ \alpha \cdot SRTT + (1 - \alpha) \cdot RTT, & SRTT > 0 \end{cases} \tag{4.1}$$

kde RTT představuje Round Trip Time, nebo-li aktuální čas v sekundách, který potřeboval paket na vykonání cesty k příjemci a zpět. Výsledná hodnota $SRTT$ je výsledkem filtrace posledních hodnot RTT. Hodnota α je konstanta v rozsahu $(0, 1)$, která kontroluje, jak rychle se $SRTT$ adaptuje na změny RTT. Doporučená hodnota pro α je hodnota *0,9*. Časovač pro přeposílání ztracených paketů s spočítá podle vztahu:

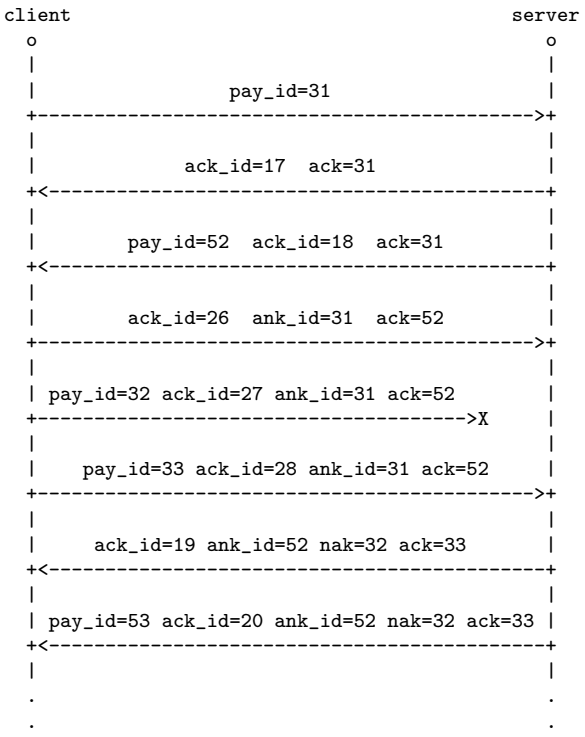
$$RTO = \beta \cdot SRTT \tag{4.2}$$

kde β je opět konstanta a její doporučená hodnota je 2. K přeposlání paketu dojde v případě, že paket není potvrzen v čase kratším jak *RTO*. Bude ukázáno, že detekce ztráty paketu na základě vypršení časovače není pro aplikace sdílené virtuální reality (ASVR) v některých případech efektivní.

Pro ASVR je někdy efektivnější použít detekci ztráty paketu na straně příjemce, když není doručen paket s očekávaným *Payload_ID*. Tento případ

detekce se aplikuje v případě, že $RTT \gg t_{FPS}$, jak je vidět na obrázku 4.33a. t_{FPS} je doba mezi dvěma snímky. Detekce ztráty paketu na základě časovače se naopak aplikuje v případě, že $RTT \ll t_{FPS}$, jak je vidět na obrázku 4.33b.

Příklad detekce ztráty paketu na straně příjemce je uveden na obrázku 4.32.

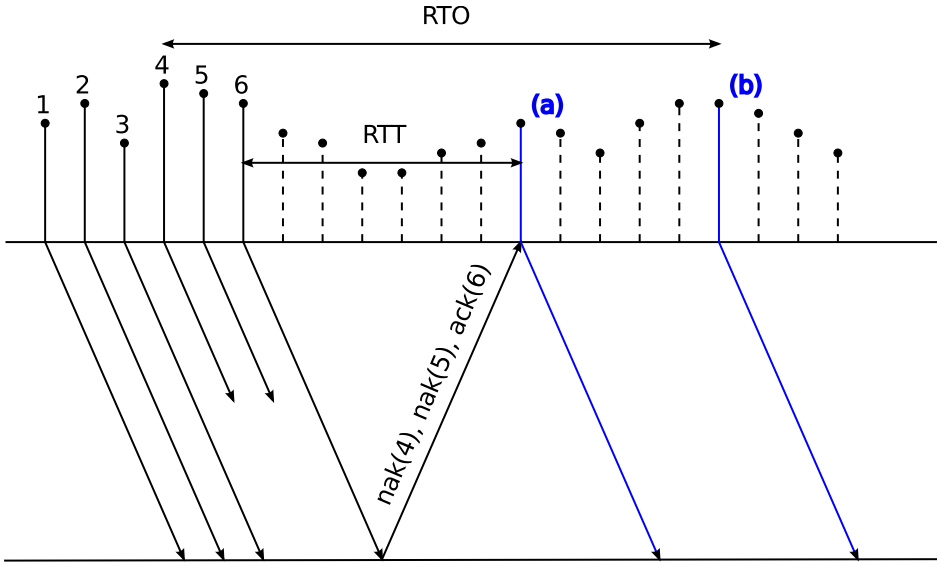


Obrázek 4.32: Příklad negativního potvrzení

V uvedeném příkladu příjemce (v tomto případě server) detekuje ztrátu payload paketu s $ID = 32$, když od odesílatele (v tomto případě klienta) obdrží místo očekávaného paketu s $ID = 32$ paket s $ID = 33$. Změnu pořadí paketu prozatím neuvažujeme. Server musí okamžitě informovat klienta o ztrátě paketu zasláním potvrzovacího paketu s příkazy $Ack = 33$ a $Nak = 32$. Tyto příkazy musí být přibalovány do všech následující paketů, dokud odesílatel nepotvrdí přijetí potvrzovacích příkazů pomocí platné hodnoty ANK v nějakém následujícím paketu.

Když klient obdrží informaci o ztrátě paketu, tak by měl ztracený paket vyvolat z historie odeslaných paketů a ze ztraceného paketu přeposlat pouze aktuální informace. Klient by neměl přeposílat například zastaralou informaci o pozici objektu z paketu $ID = 32$, když pozice objektu byla úspěšně přenesena v paketu $ID = 33$. Prosté přeposílání celého ztraceného paketu by

vedlo k nekonzistenci dat na straně klienta a serveru. Mechanismus zajišťující výběr aktuálních dat bude detailně probrán v kapitole 6.



Obrázek 4.33: Porovnání metod detekce ztráty paketu pro velké RTT

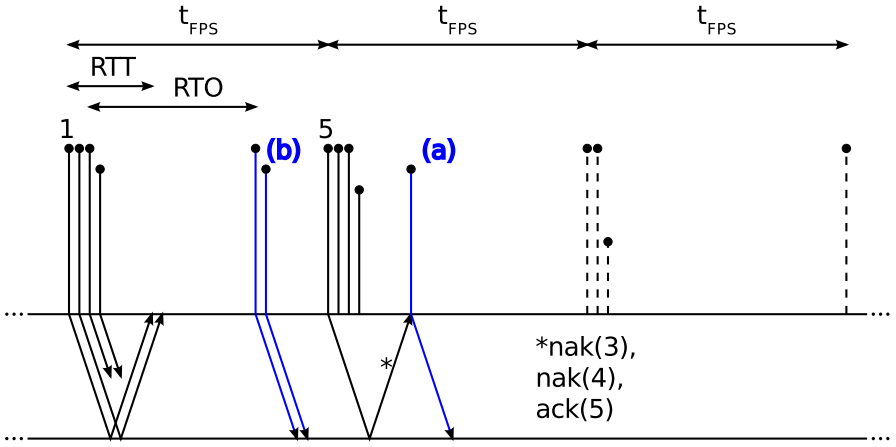
Když by byla použita pouze detekce ztráty paketu pomocí časovače na datovém okruhu jehož $SRTT = X\ s$, tak by odesílatel detekoval ztrátu paketu za $t_0 = RTO = 2 \cdot SRTT$. Za předpokladu, že by nedošlo ke ztrátě přeposlaného paketu, by příjemce obdržel paket za $t_{c1} = 2 \cdot SRTT + RTT/2$. Když je $SRTT \sim RTT$, tak výsledný čas doručení muže být aproximován pomocí:

$$t_{c1} \sim 2,5 \cdot RTT$$

Navrhovaná detekce ztráty paketu počítá s tím, že pakety se ztrácejí nejčastěji, když odesílatel posílá velké množství dat a dojde k zahlcení přenosových cest. Odesílatel musí posílat pakety v pravidelných intervalech a časový rozestup t_p mezi jednotlivými pakety musí být $t_p \ll RTT$, jak je uvedeno na obrázku 4.33. Za tohoto předpokladu příjemce detekuje ztrátu paketu za $t_0 = RTT/2 + t_p$. Jelikož příjemce musí okamžitě odeslat potvrzení o ztrátě paketu a odesílatel ztracený paket, tak zpoždění vzniklé zpracováním lze zanedbat a lze tvrdit, že příjemce obdrží přeposlaný paket za $t_{c2} = RTT/2 + t_p + RTT$. Výsledný čas doručení může být aproximován pomocí:

$$t_{c2} \sim 1,5 \cdot RTT$$

Z obrázku 4.33 je patrné, že detekce na straně příjemce *(a)* umožňuje za daných podmínek detekovat ztrátu rychleji než pomocí RTO *(b)*. Navíc ztráta všech paketů je detekovaná najednou a nikoliv postupně jako je tomu v případě detekce pomocí RTO.



Obrázek 4.34: Porovnání metod detekce ztráty paketu pro malé RTT

Na druhou stranu může být i $RTT \ll t_{FPS}$, což je uvedeno v příkladě na obrázku 4.34. V takovém případě může být značně neefektivní detekovat ztrátu paketu na straně příjemce, protože odesílatel může detekovat ztrátu paketu pomocí RTO *(b)* mnohem dříve než to umožňuje původní metoda *(a)*. Klientská aplikace totiž v mnoha případech nezasílá pakety v pravidelných intervalech, ale pakety jsou zasílány v krátkých shlucích a časové rozestupy mezi shluky t_{FPS} odpovídají FPS používané danou aplikací. Klientská aplikace navíc často nemá potřebu posílat více jak jeden paket za t_{FPS}.

Obě navrhované metody mohou fungovat současně a vždy se uplatní metoda, která detekuje ztrátu paketu dříve. Duplicitnímu přeposlání zabraňuje skutečnost, že při detekci ztráty paketu je ztracený paket odstraněn z historie. Když odesílatel obdrží příkaz *Nak* obsahující *ID*, které se nenachází v historii odeslaných paketů (ztráta paketu byla detekována časovačem, nebo odesílatel již obdržel tento příkaz *Nak*), tak tento příkaz musí ignorovat.

4.8.3 Keep-alive pakety

Když mezi klientem a serverem není potřeba přenášet žádná data, tak obě strany spojení musí každé 2 sekundy posílat keep-alive pakety, což jsou prázdné payload pakety. Když odesílatel neobdrží potvrzení o doručení Keep-

alive paketu do RTO, tak ho nesmí přeposílat, protože prázdný Keep-alive paket neobsahoval žádná užitečná data, která by bylo nutné přeposílat.

Když odesílatel neobdrží žádný paket od příjemce po dobu 30 sekund, tak je spojení s příjemcem považováno za ztracené a odesílatel by měl spojení okamžitě zavřít bez pokusu o přátelské ukončení spojení.

4.8.4 Komprese potvrzovacích příkazů

Přestože odesílatel informuje příjemce, že obdržel potvrzení o přijetí paketu pomocí *ANK_ID*, tak při velkém zpoždění na dané lince a velké frekvenci odesílání payload paketů může dojít k brzkému zaplnění velké části paketu potvrzovacími příkazy. Uvažujme následující příklad, kdy klientská aplikace odesílá každých $t_{FPS} = 16,7\ ms$ ($\sim$ 60 FPS) 10 paketů na datovém okruhu s $SRTT = 100\ ms$, by potvrzovací příkazy zabraly v paketu $(SRTT/t_{FPS}) * 10 * 5\ B \sim 300\ B$.

Libovolnou sekvenci potvrzovacích příkazů

$$\begin{array}{l} Ack(N), Ack(N+1), \ldots Ack(N+n_1), \\ Nak(N+n_1+1), Nak(N+n_1+2), \ldots Nak(N+n_2), \\ \ldots \\ Ack(N+n_{m-1}+1), Ack(N+n_{m-1}+2), \ldots Ack(N+n_m) \end{array} \tag{4.3}$$

lze rozdělit na několik subsekvencí obsahující pouze *Ack* příkazy:

$$AckSeq_i = \{Ack_0(N_i), \ldots, Ack_{n_i}(N_i + n_i)\} \tag{4.4}$$

a *Nak* příkazy:

$$NakSeq_i = \{Nak_0(N_i), \ldots, Nak_{n_i}(N_i + n_i)\} \tag{4.5}$$

, kde $n_i + 1$ je počet příkazů v dané subsekvenci.

Protože čísla pozitivně nebo negativně potvrzených payload paketů jsou v každé subsekvenci konstantně zvyšována o jedna:

$$\begin{array}{l} \ldots, Ack_j(N_j), Ack_{j+1}(N_j+1), \ldots;\ \forall j \in \langle 0, n_i - 1\rangle \\ \ldots, Nak_j(N_j), Nak_{j+1}(N_j+1), \ldots;\ \forall j \in \langle 0, n_i - 1\rangle \end{array} \tag{4.6}$$

Navíc mezi všemi subsekvencemi platí:

$$\begin{array}{l} \ldots, Ack_{n_i}(N_j), Nak_0(N_j+1), \ldots \\ \ldots, Nak_{n_i}(N_j), Ack_0(N_j+1), \ldots \end{array} \tag{4.7}$$

kde Ack_{n_i} a Nak_{n_i} jsou poslední příkazy každé subsekvence.

Původní sekvenci 4.8.4 obsahující m subsekvencí je tudíž možné zkomprimovat na sekvenci:

$$\begin{aligned} &Ack_0(N_0), Nak_0(N_1), Ack_0(N_2), \ldots \\ &\ldots Ack_0(N_{m-1}), Ack_{n_{m-1}}(N_{m-1} + n_{m-1}), \end{aligned} \tag{4.8}$$

kdy v komprimované sekvenci je z každé subsekvence pouze první příkaz *Ack* nebo *Nak*. Celá komprimovaná sekvence musí být ukončena posledním *Ack* příkazem z poslední subsekvence.

Navržená metoda umožňuje efektivně komprimovat sekvenci potvrzovacích příkazů pro velké SRTT jak je ukázáno na následujícím příkladu původní sekvence:

$$\begin{aligned} &Ack(31), Ack(32), Nak(33), Ack(34), Ack(35), Ack(36), Nak(37), Nak(38), \\ &Nak(39), Nak(40), Ack(41), Ack(42), Ack(43), Ack(44), Ack(45), Ack(46), \end{aligned}$$

kterou lze zkomprimovat na:

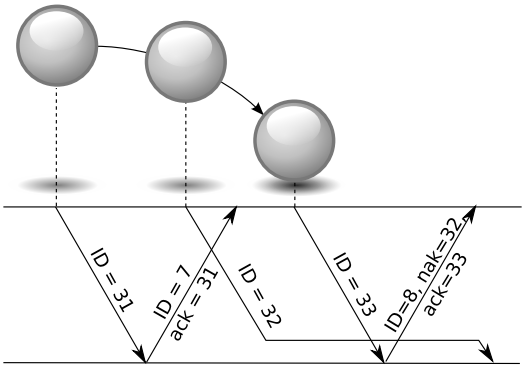
$$Ack(31), Nak(33), Ack(34), Nak(37), Ack(41), Ack(46)$$

Pro nejhorší možný případ, kdy se neustále střídají příkazy *Ack* a *Nak* je komprimovaná sekvence totožná s původní sekvencí. Navržená metoda zároveň umožňuje sekvenci potvrzovacích příkazů mít uloženou na straně příjemce přímo v komprimovaném tvaru. Algoritmus pro přidávání a redukci komprimované sekvence potvrzovacích příkazů bude uveden v kapitole 6.

4.8.5 Změna pořadí paketů

Původní protokol uvažoval, že na internetu dochází ke změně paketů velmi zřídka a tudíž je možné pakety doručené ve změněném pořadí zahazovat. Benett, Partridge a Shectman ve svém článku [5] prokázali, že změna pořadí paketů je relativně běžný jev se kterým je na internetu nutné počítat.

Původní protokol doporučuje paket doručený ve změněném pořadí zahodit, čímž se ovšem může příjemce ochudit o cenná data. Když příjemce používající nový protokol obdrží paket doručený ve změněném pořadí, tak je opět možné a samozřejmě jednodušší paket zahodit, ale příjemce se může pokusit z paketu získat informace, které jsou stále aktuální. Aktuálnost doručených dat je ovšem netriviální určit a vyžaduje to ukládat si například ID payload paketu, který daná data doručil. Z příkladu na obrázku 4.35 je patrné, že použití všech informací z payload paketu s *ID=32* by vedlo k nekonzistenci mezi příjemcem a odesílatelem. Pozice kuličky by se vrátila na straně příjemce do polohy z předchozího kroku.



Obrázek 4.35: Příklad změny pořadí paketů

Ze zpožděného paketu je ovšem možné použít data, která přijmutím payload paketu s *IP=33* nepozbyla platnosti.

Při častém výskytu změny pořadí paketu může detekce ztráty paketu na straně příjemce vést až k neefektivnímu využití přenosových cest, protože příjemce signalizuje odesílateli ztrátu dat, která byla nakonec doručena a odesílatel je nucen je znovu přeposlat. Možným řešením je pozdržet potvrzení o přijetí paketu s *ID=N* a ztrátě paketů $ID = N-1, ID = N-2, \ldots, ID = N - n_i$ po dobu t_r. Když během této doby dorazí alespoň některé pakety považované za ztracené, tak je potřeba pozměnit odpovídajícím způsobem sekvenci potvrzovacích paketů.

Hodnota t_r by měla být dostatečně malá, aby negativně neovlivňovala zpoždění doručení příkazů *Nak* pro skutečně ztracené pakety. Zároveň je potřeba upravovat hodnotu t_r aktuálním podmínkám spojení. Výchozí hodnota by měla být nulová a při přijetí payload paketu s *ID=N-i* se změněným pořadím je nutné určit čas zpoždění t_d vůči payload paketu s *ID=N*. Výsledná hodnota by nikdy neměla být větší jak

$$t_{r_{MAX}} = MIN(SRTT/30, t_{FPS}/30) \tag{4.9}$$

Příjemce by měl mít informaci o t_{FPS} díky dohadování o FPS na daném spojení, které je popsáno v části 4.5.3. Pokud odesílatel tuto informaci nemá, tak není možné korektně určit $t_{r_{MAX}}$ a t_r by mělo být nastaveno na hodnotu nula. V opačném případě se výsledná hodnota t_r spočítá podle vztahu:

$$t_r = \begin{cases} 0 & t_r = 0 \wedge t_d > t_{r_{MAX}} \\ t_d & t_r = 0 \wedge t_d \leq t_{r_{MAX}} \\ \alpha \cdot t_d + (1-\alpha) \cdot t_r & t_r > 0 \wedge t_d \leq t_{r_{MAX}}, \end{cases} \tag{4.10}$$

kde hodnota α určuje, jak příjemce rychle reaguje na změny hodnoty t_d. Doporučená hodnota pro α je 0,9. Ze vztahu 4.8.5 je patrné, že hodnota t_r nemůže přerůst určitou mez. Navíc by měl příjemce hodnotu t_r snížit o nějaké Δt s každým paketem, který byl skutečně ztracen. Paket je pro tento algoritmus považován za ztracený, pokud nedojde k jeho obdržení v čase menším jak $t_{r_{MAX}}$.

Kromě změny payload paketů může dojít i ke změně potvrzovacích paketů. Tato změna pořadí paketů ovšem nepředstavuje výraznější problém pro fungování protokolu a ani není potřeba ji řešit nějakým speciálním způsobem. Když dorazí dříve potvrzovací paket s *Ack_ID=N* jak paket s *Ack_ID=N-i*, tak novější potvrzovací paket bude obsahovat stejnou nebo delší sekvenci potvrzovacích příkazů, které budou mít za následek odstranění příslušných paketů z historie odeslaných paketů. Pozdější přijetí paketu s *Ack_ID=N-i* bude mít za následek, že už nebude nutné nic odstraňovat z historie potvrzovacích paketů.

4.9 Ukončení spojení

Ukončení datagramového spojení může být vyvoláno klientem i serverem. Nejprve bude popsána varianta, kdy ukončení spojení vyvolá svým požadavkem klient. V takovém případě je vhodné, aby se klient před ukončením spojení odhlásil od všech sdílených dat a teprve potom se pokusil spojení ukončit. Když chce klient ukončit spojení, tak se přepne do stavu *CLOSING* a serveru pošle payload paket s *ID=N*, který bude mít kromě příznaku *PAY* nastaven i příznak *FIN*. Tento payload paket může zároveň obsahovat i mít v sobě přibalený i potvrzovací paket potvrzující přijetí payload paketů ze stavu *OPEN*.

Když se klient před přepnutím do stavu *CLOSING* neodhlásí od všech sdílených dat, tak riskuje, že bude od serveru dostávat stále payload pakety, které by měl ovšem potvrzovat pomocí potvrzovacího paketu. Tento potvrzovací paket musí být zároveň přibalen do prázdného payload paketu s *ID=N* s nastaveným příznakem *FIN*. Je důležité, aby byl příznak *FIN* vždy nastaven zároveň i s příznakem *PAY*, protože potom je možné přijetí takového paketu potvrdit.

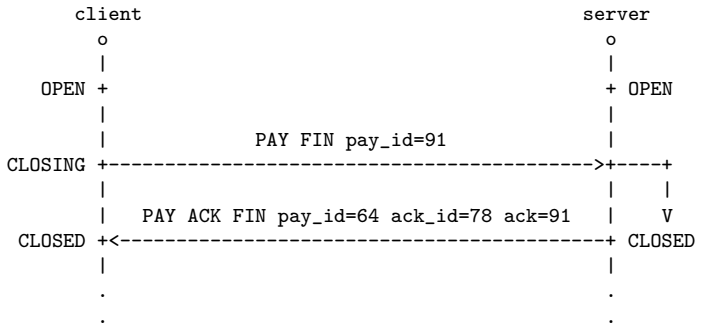
Když server obdrží payload paket obsahujíc příznak *FIN*, tak se přepne do stavu *CLOSED* a klienta automaticky odhlásí od všech dat, pokud to už neučinil sám. Odpověď serveru by měla být opět payload paket s nastaveným příznakem *FIN*, který přibaluje i potvrzovací paket s potvrzením o přijetí payload paketu s *ID=N*.

Když se klient přepne do stavu *CLOSING* a neobdrží odpověď od ser-

veru do 1 sekundy, tak by měl klient poslat další odhlašovací payload paket s totožným *ID=N*. Každá další hodnota časovače, by se měla nastavovat pomocí stejného algoritmu jako byl popsán na straně 52 pro stav *REQUEST*. Když klient neobdrží odpověď na ukončovací paket do 30 sekund od přepnutí do stavu *CLOSING*, tak by měl spojení automaticky zavřít a komunikaci se serverem ukončit.

Server by měl setrvat ve stavu *CLOSED* po dobu 30 sekund, protože během této doby musí stále odpovídat na případné klientovy požadavky na uzavření spojení.

Příklad uzavření spojení vyvolaný klientem je uvedený na obrázku 4.36.



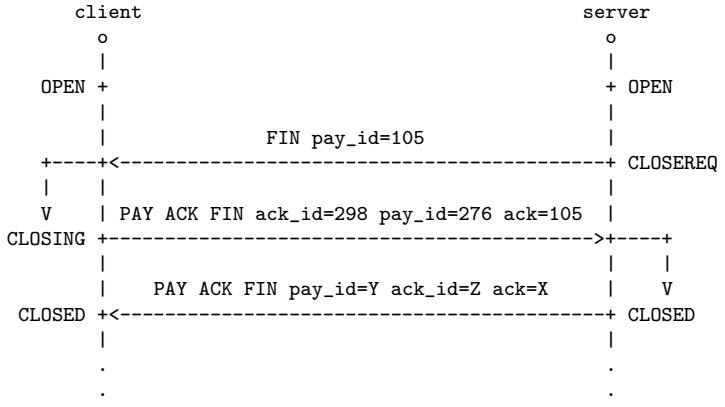
Obrázek 4.36: Příklad ukončení datagramového spojení

Když se klient korektně přepne do stavu *CLOSED* po odpovědi od serveru a datagramové spojení bylo zabezpečeno pomocí DTLS, tak by měl klient inicializovat i ukončení DTLS.

Když ukončení spojení chce vyvolat server, tak se přepne do stavu *CLOSEREQ* a pošle payload paket obsahující nastavený příznak *FIN*. Stav *CLOSEREQ* se principiálně neliší od stavu *OPEN* pouze jsou ve všech jeho payload paketech nastavené i příznaky *FIN*. Po přepnutí do stavu *CLOSEREQ* server očekává, že se klient odhlásí od všech svých dat a přepne se do stavu *CLOSING* ve kterém začne posílat payload pakety s příznakem *FIN*. Následně komunikace mezi klientem a serverem probíhá stejně jako za situace, kdy ukončení spojení vyvolal klient.

Když klient nereaguje na požadavek serveru o ukončení spojení a do 30 sekund od přepnutí do stavu *CLOSEREQ* nepošle serveru payload paket s příznakem *FIN*, tak by měl server komunikaci s klientem násilně ukončit.

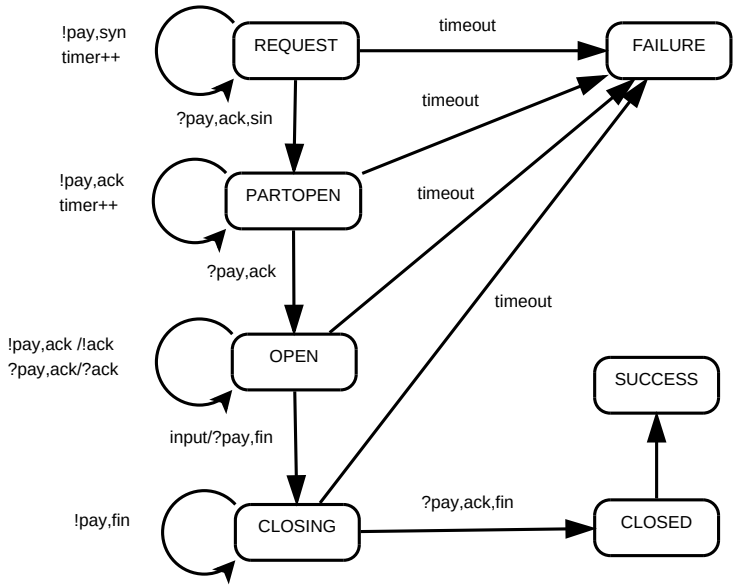
Příklad ukončení spojení vyvolané serverem je uveden na obrázku 4.37.



Obrázek 4.37: Příklad ukončení datagramového spojení

4.10 Verifikace datagramového spojení

Verifikační model byl vytvořen pro celé datagramové spojení a je uveden v příloze na straně 137. Verifikační model nebyl rozdělen na více částí, protože jednotlivé stavy spojení spolu úzce souvisí. Stavový model klienta je uvedený na obrázku 4.38 a stavový model serveru je uvedený na obrázku 4.39.



Obrázek 4.38: Stavový model datagramového spojení klienta

Cílem verifikačního modelu bylo ověřit, že navržený handshake, ukončení spojení a resend mechanismus jsou navržené korektně a že neobsahují žádný

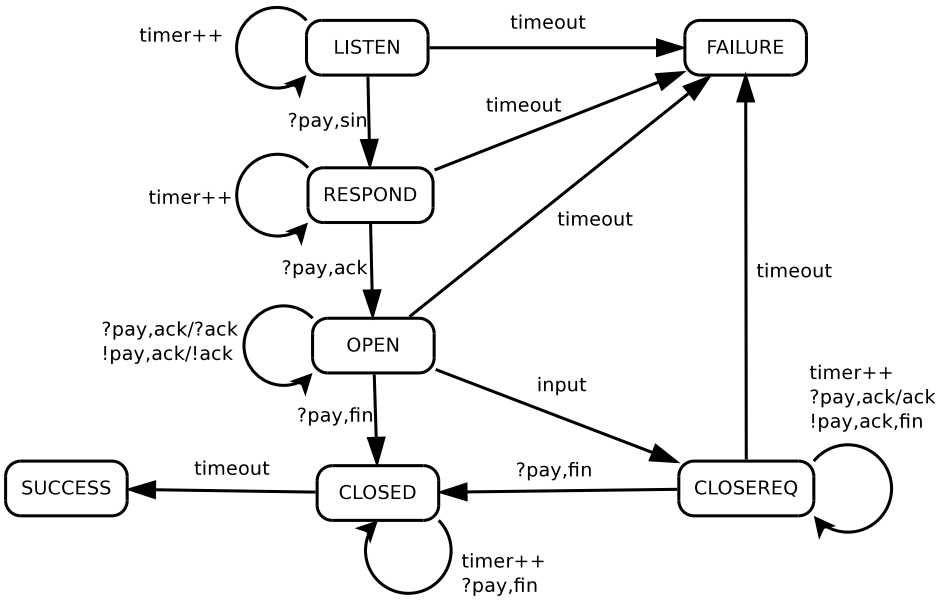
deadlock nebo nevyvíjející se cyklus. Oba procesy (klient a server) spolu komunikují pomocí dvou kanálů:

```
/*                                PAY,  ACK,  SYN,  FIN,  PAY_ID  ACK_ID  ACK_MSG */
chan q1 = [QLEN] of {bit, bit, bit, bit, short, short, short};
chan q2 = [QLEN] of {bit, bit, bit, bit, short, short, short};
```

které umožňují přenášet zjednodušenou hlavičku paketu a jednu potvrzovací zprávu. Každý paket obsahuje z původní sekvence potvrzovacích příkazů pouze poslední příkaz *Ack*. Z tohoto důvodu je z hlavičky vypuštěn *ANK* příznak a *ANK_ID*. Použití jednoho potvrzovacího příkazu je pro verifikaci dostatečné, protože během handshaku se používá pouze jeden potvrzovací příkaz *Ack* a cílem verifikace resend mechanismu bylo prokázat, že je schopný doručit poslední payload paket $ID = N$.

Oba komunikační kanály mají v tomto případě nenulovou délku, která byla nastavena na hodnotu 3. Verifikace proběhla korektně i s větší délkou komunikačního kanálu, ale za cenu větších paměťových nároků. Pro korektní provedení verifikace bylo nutné nastavit, aby se z odeslání zprávy nestala blokující operace v případě, že komunikační kanál byl plný, ale aby došlo k zahození takové zprávy.

Výsledky verifikace datagramového spojení (uvedeny v příloze na straně 145) prokázaly, že se v této části protokolu nenachází žádný deadlock, nevyvíjející se cykly ani nekorektní koncové stavy.



Obrázek 4.39: Stavový model datagramového spojení serveru

4.11 Flow Control

Flow Control (FC) zabraňuje rychlému odesílateli, aby zahltil pomalého příjemce. Odesílatel by neměl posílat více dat než mu stanovuje tzv. posuvné okénko, jehož koncept vychází z TCP. Velikost tohoto posuvného okénka *swin* se určí pomocí

$$swin = min\{rwin, cwin\} \tag{4.11}$$

kde *rwin* je počet paketů plné velikosti, které je příjemce ochoten přijmout a *cwin* je Congestion Window, jež bude popsáno v následující části. Odesílatel by neměl odeslat více nepotvrzených paketů než mu dovoluje hodnota *swin*. Naopak příjemce v každém payload i potvrzovacím paketu deklaruje aktuální velikost *rwin*. Velikost deklarovaného okénka by měla být určená dostupným místem ve vstupní frontě daného spojení. Vstupní a výstupní fronty spojení budou popsány v části 6.

4.12 Congestion Control

Congestion Control (CC) je mechanismus, který by měl předcházet a pomáhat zlepšovat zahlcení přenosových cest. Mechanismus pro protokol Verse by měl vycházet z CC protokolu TCP, jak je popsaný v [33] a [2], ale jeho chování by mělo být upraveno vlastnostem a potřebám protokolu Verse.

V současné době není žádný skutečný algoritmus CC pro protokol Verse ani implementovaný ani navržený. Klient se serverem se mohou dohodnout během handshaku na použití *CC_NONE*, kdy *cwin* ze vztahu 4.11 je vždy rovno 2^{16}. Navržení vhodného algoritmu pro CC je dalším cílem vývoje protokolu Verse.

Kapitola 5

Datový model

Aby byla možná komunikace mezi servere a různými klienty, tak všichni musí vycházet ze stejného datového modelu. Tento datový model ovlivňuje hlavně strukturu paketů pro přenos užitečných dat. Vlastní uložení dat na straně klienta specifikace neřeší. Data jsou strukturována do uzlů podobně jako tomu bylo v původním protokolu Verse. Hlavní rozdíl proti původnímu datovému modelu spočívá v tom, že existuje pouze jeden obecný typ uzlu a uzly musí být umístěny do stromové struktury. Nový protokol dále obsahuje podporu pro přístupová práva na úrovni uzlů, možnost dočasně zamykat uzly a každému uzlu je možné přiřadit určitou prioritu. Podobně jako v původním protokolu může každý uzel obsahovat skupinu tagů, které by měly sloužit především pro popis dat uložených v tzv. vrstvách daného uzlu. Tagy mohou kromě popisu vrstev sloužit i k uložení dat. Každý uzel může reprezentovat objekt, geometrii objektu, materiál, texturu, animační křivku, parametrickou plochu, atd. Výsledná sdílená 3D scéna může být vytvořena pomocí vhodných vazeb a referencí mezi jednotlivými uzly.

5.1 Uživatelé a uživatelské účty

Ve specifikaci původního protokolu byl uživatelský účet použit pouze pro ověření uživatele pomocí jeho jména a hesla. Po autentizaci mohl uživatel provádět cokoliv se všemi sdílenými daty.

V novém protokolu má každý uživatel přiřazen kromě jedinečného uživatelského jména i jedinečný číselný identifikátor (User ID), který může nabývat hodnot v rozmezí $\langle 100, 65534 \rangle$. Rozsah $\langle 100, 999 \rangle$ je určen privilegovaným uživatelům. Server by měl autentizovat pouze uživatele z rozsahu $\langle 1000, 65534 \rangle$. Hodnota 65535 má speciální význam pro nastavování přístupových práv a nikdy by neměla být přiřazena konkrétnímu uživateli.

Uživatel reprezentující samotný server má *User ID* rovný hodnotě 100. Další hodnoty v rozsahu $\langle 100, 999 \rangle$ jsou rezervovány pro privilegované uživatele slave serverů distribuované variantu verse serveru, která byla nastíněna na straně 44.

Verse klient vždy zná své *User ID*, pod kterým je přihlášený, z příkazu *UserAuthSucces*, jež je uvedený na straně 35.

5.2 Uzly

Aby bylo možné jednoznačně identifikovat kterýkoliv sdílený uzel, tak každý uzel musí mít jedinečný identifikátor (Node ID). Musí to být server a nikoliv klienti, kdo přiděluje novým uzlům jejich *Node ID*, protože pouze server má možnost zajistit jejich jedinečnost. *Node ID* může nabývat hodnot od $\langle 0, 2^{32} - 2 \rangle$. Rozsah možných hodnot je poměrně široký, což dává možnost přiřadit některým hodnotám a rozsahům speciální význam.

Všechny uživatelské účty mají například vytvořený speciální uzel jehož *Node ID* odpovídá *User ID* daného uživatele. Tento uzel může obsahovat další informace o daném uživateli a Verse klienti mají k dispozici jednoduchý způsob jak zjistit seznam platných uživatelů a jejich *User ID*.

Původní protokol dával klientům možnost zvolit si identifikátor nově vytvářeného uzlu. Když klient zaslal příkaz pro vytvoření uzlu, který obsahoval i návrh identifikátoru a tento identifikátor již existoval, tak server nově vytvořenému uzlu přiřadil nepoužívaný identifikátor. Klient musel s tímto chováním počítat, ale dané pravidlo rozhodně nepřispívalo k jednoduché implementaci Verse klientů.

5.2.1 Přístupová práva

Přístupová práva by měla umožňovat omezit přístup k libovolným sdíleným datům na serveru. Přístupová práva jsou navržena tak, aby je mohli jednoduše nastavovat uživatelé virtuální reality. Z tohoto důvodu návrh přístupových práv neobsahuje koncept skupin uživatelů, jak je používají UNIXové souborové systémy. Koncept přístupových práv použitý v novém protokolu Verse více vychází z Access Control List (ACL).

Vlastník

Každý uzel má svého vlastníka, který má právo daný uzel číst a zapisovat do něj, což znamená, že se může přihlásit k přijímání změn provedených s daty uvnitř tohoto uzlu a sám může měnit data v tomto uzlu. Vlastník uzlu se

může svého vlastnictví vzdát ve prospěch jiného uživatele. Tuto změnu může provést buď vlastník nebo privilegovaný uživatel.

Ostatní uživatelé

Každý uzel má vždy nastaveno, co s ním mohou dělat všichni ostatní uživatelé. Vlastník souboru může vybranému uživateli explicitně nastavit dodatečná přístupová práva. Může mu povolit číst a zapisovat do daného uzlu.

Příkazy pro změnu vlastnictví a přístupových práv budou detailně vysvětleny v části 5.2.8 a 5.2.9.

5.2.2 Stromová struktura

Uspořádání uzlů do obecného grafu bylo v původním protokolu volitelné. Vazby mezi jednotlivými uzly bylo možné vytvořit pouze na základě explicitního požadavku klienta. V novém protokolu je vyžadováno, aby byly uzly uspořádány do stromové struktury. Uspořádání uzlů do stromové struktury zjednodušuje některé operace jako je přihlášení k datům, nastavení priorit apod. Není například nutné explicitně nastavit prioritu pro každý uzel, ale stačí nastavit prioritu pro jeden uzel a tato priorita se dědí na všechny uzly v dané větvi.

Stromová struktura musí vždy sledovat určitá pravidla. Prvním z nich je, že kořenem stromové struktury musí být uzel s ID=0. Jeho vlastníkem musí být server a nikdo další by neměl mít právo do tohoto uzlu zapisovat, pouze ho číst. Kořenový uzel musí mít vždy tři potomky:

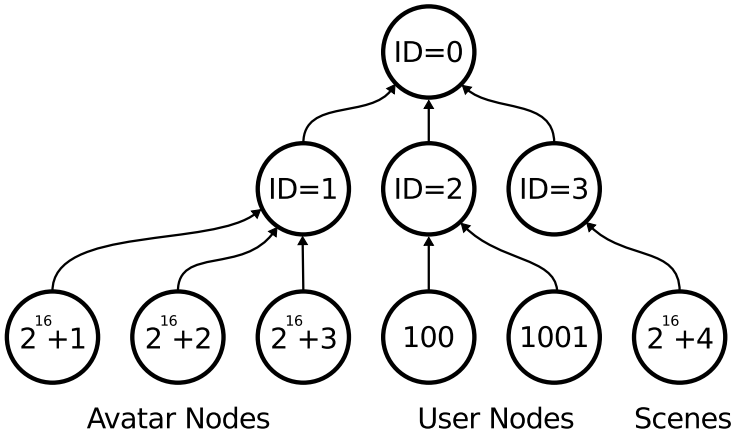
- *Node ID = 1* je předek všech uzlů reprezentujících avatary
- *Node ID = 2* je předek všech uzlů reprezentujících uživatele
- *Node ID = 3* je předek pro kořenové uzly jednotlivých 3D scén

Vlastníkem předka všech avatarů je opět server a nikdo další do něj nesmí mít právo zapisovat. Ostatní uživatelé musí mít právo tento uzel číst. Server musí vytvářet uzly reprezentující avatary jako potomky uzlu s ID=1 a zároveň by zde neměl vytvářet jiné uzly. Díky tomu jsou klienti schopni určit seznam klientů přihlášených k danému serveru.

Podobné vlastnosti platí i pro předka všech uzlů reprezentujících uživatele. Tento uzel je ve vlastnictví serveru a ostatní uživatelé tento uzel smí pouze číst. Potomky tohoto uzlu smí být pouze uzly reprezentující platné uživatele, nejenom aktuálně přihlášené uživatele. Ostatní klienti musí mít šanci určit

User ID i uživatelů, kteří nejsou momentálně přihlášení. Vlastníkem uzlů reprezentujících uživatele musí být opět server. Uživatel, jehož *User ID* je rovno danému *Node ID*, by měl mít právo do daného uzlu zapisovat. Ostatní uživatelé by měli mít pouze právo číst obsah tohoto uzlu.

Vlastníkem posledního potomka kořenového uzlu s ID=3 je opět server. Tento uzel smí číst a zapisovat do něj všichni uživatelé. Ve větvích vycházejících z uzlu s ID=3 by mělo docházet k vlastnímu sdílení dat mezi jednotlivými klienty. Na obrázku 5.1 je uveden příklad uspořádíní uzlů do jednoduché stromové struktury.



Obrázek 5.1: Příklad stromové struktury

Uživatel by měl mít možnost změnit uzlu jeho rodiče. Změna rodiče je uživateli dovolena, pokud mu uzel patří a zároveň musí mít právo zapisovat do rodičovského uzlu.

Při změně rodičovského uzlu musí server hlídat, aby nedošlo k rozdělení stromové struktury na dva nesouvislé grafy (jeden nový strom a graf obsahující kružnicí). Tuto podmínku může server jednoduše zajistit tím, že nedovolí, aby se novým rodičem uzlu stal jakýkoliv jeho následovník.

5.2.3 Avatar

Každá 3D aplikace používá koncept tzv. avatara. Avatar je reprezentant uživatele ve virtuální realitě. Má vždy přiřazenou minimálně jednu virtuální kameru, kterou uživatel sleduje obsah virtuální reality. Dále může mít avatar definované rozměry zabraňující průchod příliš malými otvory, hmotnost, koeficient tření apod.

Každý Verse klient musí mít svého avatara. Je vhodné, aby Verse klient sdílel na serveru informace o svém avataru. U každého typu aplikace SVR

je vhodné sdílet o avatarovi rozdílnou sadu informací. Některé aplikace avatara buď vůbec nevyžadují zobrazovat (např. 3D modelační aplikace) nebo zobrazují pouze jeho kameru, aby měli ostatní účastníci ponětí o tom, kde se daný uživatel nachází a na čem pravděpodobně pracuje. Při propojení CAVE [10] pracovišť je zase žádoucí, aby byl avatar pro ostatní účastníky viditelný pomocí animované postavy. Ve všech případech je ovšem žádoucí, aby měli účastníci možnost jednoznačně určit, kolik Verse klientů je k serveru přihlášeno a pod jakými uživateli.

Server tudíž po autentizaci uživatele vytvoří nový uzel označovaný jako *Avatar Node*. Jeho ID se klient dozví v příkazu *UserAuthSucces*, jenž je uvedený na straně 35. Specifikace nic neříká o tom, jakým způsobem se mají sdílet informace o avatarovi v tomto uzlu, protože každý typ aplikace vyžaduje jiný přístup. Toto bude kladeno na bedra DED.

Specifikace ovšem má několik základních pravidel pro *Avatar Node*. Jeho vlastníkem je vždy server a server musí zajistit existenci tohoto uzlu po celou dobu spojení klienta s Verse serverem. Po ukončení spojení by měl server tento uzel smazat včetně všech jeho potomků. Nadřazeným uzlem *Avatar Node* je jak už bylo řečeno uzel s ID=1. Server jako vlastník může daný uzel číst a může do něj zapisovat. Stejné práva by měl mít i uživatel pod kterým je přihlášený daný Verse klient. Ostatní uživatelé by měli mít možnost daný uzel pouze číst.

Uzly pro avatary nemají vyhrazený žádný speciální rozsah. *Node ID* jsou jim přiřazovány jako běžným uzlům z rozsahu $\langle 0, 2^{32} - 2\rangle$.

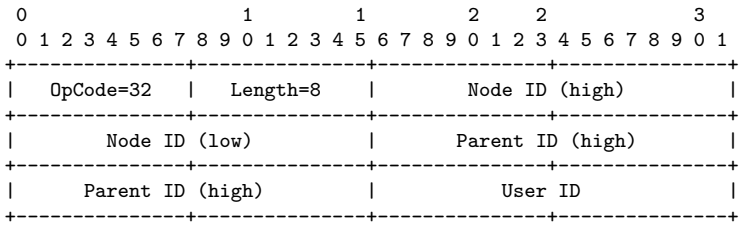
5.2.4 Základní operace s uzly

Doposud uváděné příkazy protokolu Verse byly tzv. systémové příkazy, které ovlivňovaly vlastnosti celého spojení. Uzlové příkazy naopak slouží pro přenos užitečných dat. Každý uzlový příkaz má dvě části; adresu, která identifikuje měněnou entitu, a vlastní data. Kompletní seznam všech uzlových příkazů včetně popisu jednotlivých položek je uveden v příloze na straně 147.

Vytvoření nového uzlu

Příkaz pro vytvoření nového uzlu má strukturu uvedenou na obrázku 5.2. Jednotlivé položky příkazu mají následující význam:

- *OpCode=32* - položka určující, že se jedná o příkaz *'Node Create'*
- *Length* - délka příkazu v bytech. Minimální délka je 8 bytů.



Obrázek 5.2: Struktura příkazu pro vytvoření nového uzlu

- *Node ID* - identifikátor nového uzlu.
- *Parent ID* - identifikátor nadřazeného uzlu.
- *User ID* - vlastník nového uzlu.

Výše uvedený příkaz posílá jak server tak klient. Server posílá příkaz *Node Create* klientům, když je vytvořen na serveru nový příkaz a tito klienti jsou přihlášeni k nadřazenému uzlu. Dále server pošle tento příkaz klientovi v případě, že se klient přihlásí k uzlu, který má nějaké podřízené uzly.

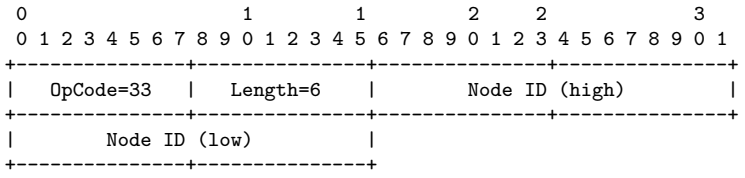
Klient může také poslat tento příkaz, ale musí dodržovat následující pravidla. Položka *Node ID* musí být vyplněna samými jedničkami ($2^{32}-1$). Položka *Parent ID* musí být totožná s identifikátorem uzlu, který reprezentuje avatara daného klienta. Konečně hodnota *User ID* se musí shodovat s identifikátorem uživatele pod kterým je uživatel pomocí tohoto klienta přihlášen k serveru. Příkaz *Node Create* v případě klienta slouží jako žádost o vytvoření nového uzlu.

Když má server dost systémových prostředků, tak by měl na klientův požadavek vytvořit nový uzel s jedinečným *Node ID*, jako jeho nadřazený uzel nastavit uzel jeho avatara a vlastníkem by měl být daný uživatel. Klient je při obdržení daného příkaz schopen určit z *Parent ID* a *User ID*, jestli je to uzel o jehož vytvoření žádal. To, že je u nově vytvořeného uzlu *Parent ID* roven *Avatar ID* má další výhodu v tom, že je zaručena existence nadřazeného uzlu pro všechny nově vytvořené uzly. Uzel reprezentující avatara může zrušit jenom server po té, co je ukončena komunikace klienta se serverem.

Zrušení existujícího uzlu

Při mazání uzlů se musí dodržovat pravidlo, že uzel může smazat pouze jeho vlastník. Pokud se má smazat uzel V_i, který má alespoň jednoho potomka, tak lze uvažovat o rekurzivním mazání všech větví vycházejících z uzlu V_i. Tento přístup má úskalí v tom, že v některé větvi se může nacházet uzel

V_j, který nemá stejného vlastníka jako uzel V_i. V takovém případě nelze smazat ani uzel V_j, ani žádný uzel na cestě mezi uzly V_i a V_j. Server smí smazat pouze uzly, které má daný uživatel právo smazat a zároveň takový uzel nesmí být předchůdcem uzlu, který daný uživatel nemá právo smazat. Příkaz *Node Destroy* pro smazání uzlu je uveden na obrázku 5.3.



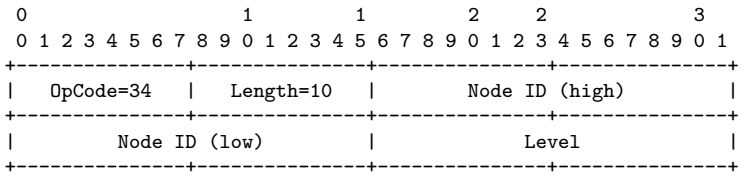
Obrázek 5.3: Struktura příkazu pro zrušení uzlu

Server může klientovi poslat příkaz *Node Destroy* teprve v případě, že od klienta obdržel potvrzení o doručení paketu s odpovídajícím příkazem *Node Create*. Kdyby server nedodržoval toto pravidlo, tak by mohlo dojít k nekonzistenci dat na straně klienta a serveru.

5.2.5 Přihlašování k uzlům

I původní protokol Verse pracoval s koncepcí přihlašování se k uzlům a jejich datům, která umožňuje klientovi kontrolovat jaká data mu bude server zasílat. Tuto koncepci v upravené podobě přebral i nový protokol Verse.

Když se klient přihlásí k Verse serveru a přepne se na datagramovém spojení do stavu *OPEN*, tak mu server nezačne automaticky posílat všechna sdílená data. Klient si musí explicitně o data říci pomocí tzv. přihlášení se k datům. Nejprve se musí přihlásit k uzlu pomocí příkazu *Node Subscribe*, který je uveden na obrázku 5.6. Když má klient právo daný uzel číst a obsahuje nějaká data nebo potomky, tak ho o tom server informuje zasláním příslušných příkazů.



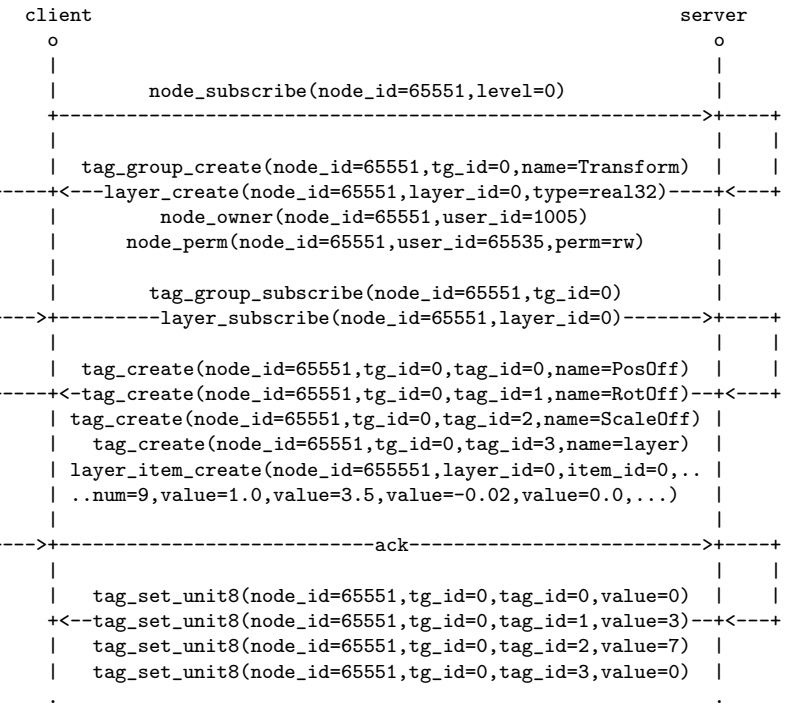
Obrázek 5.4: Struktura příkazu pro přihlášení se k uzlu

Klient by se měl nejprve přihlásit ke kořenovému uzlu jehož ID je vždy rovno nule. Zbytek stromové struktury může objevovat postupně, když se

dozví o příslušných potomcích, nebo může použít rekurzivního přihlašování. Příkaz *Node Subscribe* totiž obsahuje kromě položky *Node ID* i položku *Level*, která říká o kolik úrovní rekurzivního přihlašování se má server pokusit. Pokud má položka *Level* hodnotu 0xFFFF, značí to, že se má server pokusit o neomezené rekurzivní přihlášení. Když se klient přihlásí k uzlu, neznamená to, že mu server pošle veškerá data obsažená v tomto uzlu. Server pošle klientovi příkazy informující ho, jaké uzel obsahuje skupiny tagů a vrstvy. Teprve když klient ví, že daný uzel obsahuje nějaká data, tak se k nim může přihlásit pomocí příslušných příkazů.

Server vždy pošle klientovi po přihlášení k uzlu, kdo je jeho vlastníkem a kompletní přístupová práva k danému uzlu. Pokud by klient neměl právo daný uzel ani číst, dozví se to po přihlášení k tomuto uzlu. Zároveň se dozví, který uživatel mu tato práva upírá.

Příklad přihlášení k uzlu je uveden na obrázku 5.5.

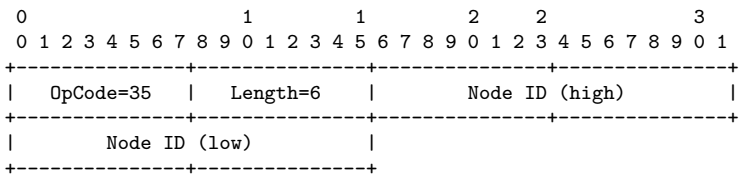


Obrázek 5.5: Zjednodušený příklad přihlášení se k datům jednoho uzlu

V tomto příkladě se klient přihlásil k uzlu *65551*. Server mu následně poslal kromě informace o vlastnictví a přístupových právech i příkazy *Tag-Group Create* a *Layer Create* (blíže popsáno v části 5.2.12 a 5.2.13). Tyto příkazy klientovi říkají, že uzel obsahuje skupinu tagů *"Transform"* a jednu vrstvu. Klient se na základě zjištěných ID přihlásil jak ke skupině tagů, tak

i k vrstvě. Server mu na tento požadavek odpověděl příkazy pro vytvoření tagů v dané skupině a zároveň mu poslal příkaz pro vytvoření všech položek ve vrstvě 0.

V případě, že už klient nemá dále zájem být informován o změnách v daném uzlu (přidání nového potomka, skupiny tagů, apod.), tak se může od daného uzlu odhlásit pomocí příkazu *Node Unsubscribe*, jehož sktruktura je uvedena na obrázku 5.6.



Obrázek 5.6: Struktura příkazu pro odhlášení se od uzlu

Klient může být přihlášen i k datům uvnitř uzlu, pak je při použití příkazu *Node Unsubscribe* odhlášen automaticky od dat uvnitř uzlu. Má-li uzel nějaké potomky, tak je klient rekurzivně odhlášen od všech jeho potomků.

5.2.6 Potvrzování uzlových příkazů

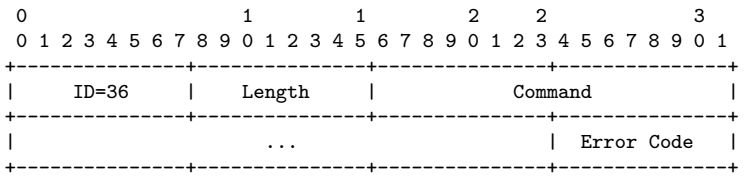
Server i klient provádějí potvrzování přijetí payload paketů, jak bylo popsáno v části 4.8. Server může provádět potvrzování i samotných příkazů, protože klient může poslat neplatný příkaz (může se odkazovat na neexistující uzel, uživatel nemá dostatečné oprávnění, apod). Když klient odešle příkaz *Node Subscribe*, tak než dojde k jeho přijetí serverem, může dojít k jeho smazání jiným klientem. Z tohoto důvodu je nutné chápat uzlové příkazy odeslané klientem pouze jako požadavky, které mohou ale nemusí být vyřízeny.

Pozitivní potvrzování příkazů

Pozitivní potvrzování spočívá v tom, že server musí přeposlat platný uzlový příkaz všem klientům, kteří jsou přihlášeni k danému uzlu nebo k příslušným datům. Předpokládá se, že klient posílající uzlový příkaz je přihlášen k danému uzlu. Klient se ovšem na pozitivní potvrzení příkazu (nikoliv paketu) nesmí spolehnout, protože než server daný příkaz stihne klientovi přeposlat, tak tento příkaz může být v odesílací frontě serveru přepsán jiným příkazem.

Ohlašování chybných příkazů

Server nemusí klienta informovat o neplatném příkazu. Toto chování je nepovinné a server může neplatný příkaz zahodit. Použití negativního potvrzování může být užitečné pro vývojáře Verse klientů, když se snaží odhalit chybné chování aplikace. Pro běžný provoz není ohlašování chybných příkazů nezbytně nutné. V případě, že server implementuje ohlašování chybných příkazů, tak by měl server informovat klienta o přijetí neplatného příkazu pomocí příkazu *Node Error*, jehož struktura je uvedena na obrázku 5.7.

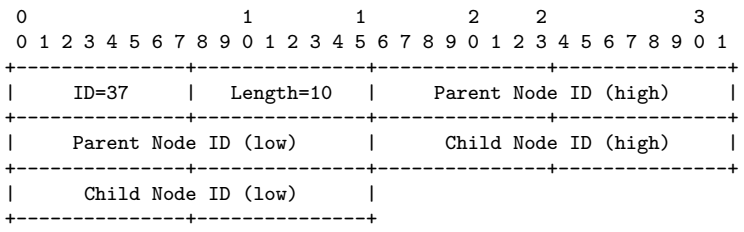


Obrázek 5.7: Struktura příkazu pro ohlášení chybného příkazu

Každý příkaz by měl mít vlastní sadu chybových kódů. Pokud příkaz zaslaný klientem obsahuje více chyb, tak server musí v příkazu *Node Error* ohlašovat první nalezenou chybu.

5.2.7 Nastavení vazeb mezi uzly

Klient má možnost změnit rodiče u vlastních uzlů pomocí příkazu *Node Link* jehož struktura je uvedena na obrázku 5.8.

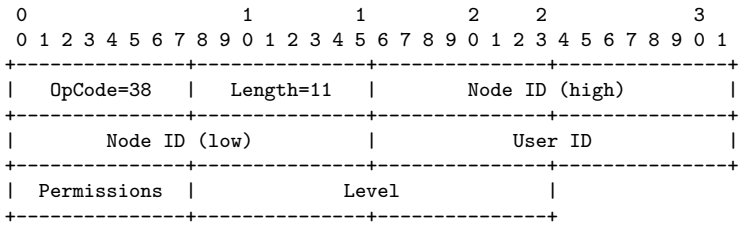


Obrázek 5.8: Struktura příkazu pro změnu vazeb mezi uzly

Klient se musí řídit pravidly, jež byla uvedena na straně 74 v části věnované stromové struktuře. Při změně rodičovského uzlu musí nový rodičovský uzel (*Parent Node ID*) existovat a klient do něj musí mít samozřejmě právo zapisovat. V případě, že *Parent Node ID* v době přijetí příkazu serverem již neexistuje, tak je tento příkaz buď ignorován nebo server odešle klientovi příkaz s ohlášením chybného příkazu.

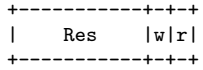
5.2.8 Změna přístupových práv

Jak již bylo řečeno, klient má právo změnit přístupová práva u uzlů, které mu patří a zároveň server musí ostatní klienty informovat o změně přístupových práv. V obou případech se používá příkaz *Node Permission*, jehož struktura je uvedena na obrázku 5.9.



Obrázek 5.9: Struktura příkazu pro změnu přístupových práv

Položka *Permissions* obsahuje jednotlivé příznaky odpovídající jednotlivým právům. Práva se nastavují pomocí příznaku *w* (uživatel může do uzlu zapisovat) a *r* (uživatel může uzel číst), jak je to uvedeno na obrázku 5.10. Zbylý prostor je použit jako rezerva pro další příznaky.

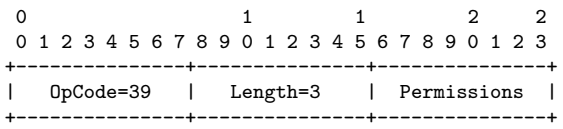


Obrázek 5.10: Bitové příznaky pro nastavení přístupových práv

V případě, že server příjme platný příkaz *Node Permission*, tak ho musí přeposlat všem klientům, kteří jsou přihlášeni k danému uzlu. Server musí ignorovat příkazy *Node Permission* od klientů, kteří nejsou vlastníci daného uzlu, případně poslat odesílateli patřičný příkaz *Node Error*.

Server nastavuje u nově vytvořených uzlů výchozí práva. Vlastník musí mít vždy právo do uzlu zapisovat a číst ho. Výchozí pravidlo je, že ostatní uživatelé mají právo nový uzel číst. Aby nemusel klient nastavovat explicitně přístupová práva pro ostatní uživatele u každého nového uzlu, tak je možné použít příkaz *Node Default Permission*, jenž je uveden na obrázku 5.11.

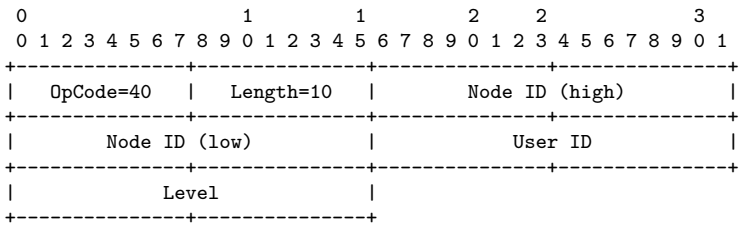
Tento příkaz umožňuje nastavit výchozí práva pro ostatní uživatele u nově vytvořených uzlů. Server potom automaticky nastavuje novým uzlům zvolená výchozí práva, které vytvořil daný klient.



Obrázek 5.11: Struktura příkazu pro nastavení výchozích přístupových práv u ostatních uživatelů

5.2.9 Změna vlastníka

Když chce vlastník uzlu předat vlastnictví uzlu jinému uživateli, tak musí použít příkaz *Node Owner* jehož struktura je uvedena na obrázku 5.12.



Obrázek 5.12: Struktura příkazu pro změnu vlastníka uzlu

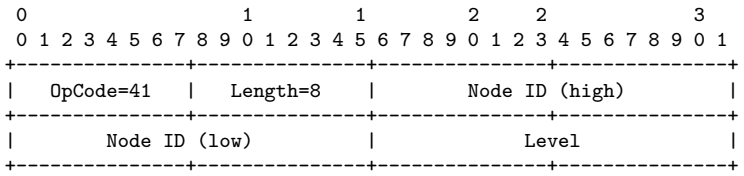
Přijme-li server tento příkaz, tak by ho měl odeslat všem klientům, kteří jsou přihlášeni k danému uzlu. Server musí ignorovat požadavky klientů, snažící se změnit vlastnictví uzlů, jenž jim nepatří. Když to server implementuje, tak může danému klientovi poslat příkaz *Node Error* s příslušným chybovým kódem.

5.2.10 Dočasné zámky

Klient by měl mít právo dočasně zamknout uzel, aby ostatní klienti nemohli tento uzel modifikovat. Dočasné uzamčení uzlu jednak umožňuje uživateli nerušeně modifikovat daný uzel a zároveň zjednodušuje implementaci protokolu Verse do existujících aplikací. Při uzamčení uzlu není nutné řešit, jestli příchozí příkazy pro daný uzel pocházejí od ostatních klientů nebo se jedná o pouhé potvrzení odeslaných příkazů.

Klient se může pokusit uzamknout uzel zasláním příkazu *Node Lock*, jenž je uveden na obrázku 5.13. V případě, že má uzel potomky, může klient specifikovat pomocí parametru *Level* kolikrát se má pokusit o rekurzivní uzamčení uzlu. Před uzamčením všech požadovaných uzlů musí server zkontrolovat, zdali je možné uzamčení provést. Pokud tato podmínka není splněna (alespoň jeden uzel je již uzamčen jiným klientem), tak server uzamčení provést

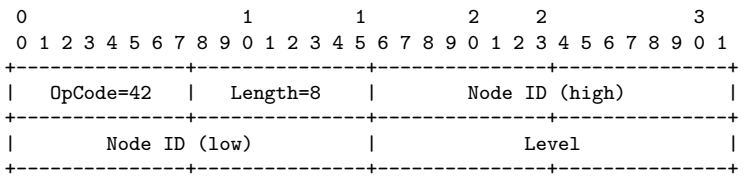
nesmí. V případě uzamčení musí server zaručit atomicitu kontroly uzamčení a vlastní zamčení daných uzlů. Za žádných okolností nesmí nastat situace, kdy jeden uzel mají uzamčen dva klienti.



Obrázek 5.13: Struktura příkazu zamknutí uzlu

Klient musí požadavky na uzamčení uzlu opakovat každých 5 sekund. Server totiž uzel automaticky odemkne po uplynutí časového limitu 30 sekund, když klient nezopakuje požadavek na jeho uzamčení.

Klient může uzel dobrovolně odemknout pomocí příkazu *Node Unlock*, jehož struktura je uvedena na obrázku 5.14.



Obrázek 5.14: Struktura příkazu odemčení uzlu

Server by měl příkaz *Node Unlock* ignorovat v případech, kdy uzel není uzamčený nebo má uzel uzamčený jiný klient.

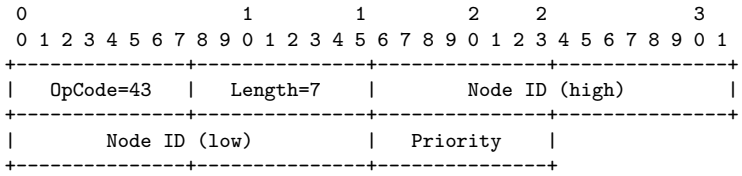
5.2.11 Priority uzlů

Uživateli virtuální reality není potřeba zobrazovat všechny objekty se stejnou úrovní detailů (LOD), protože některé objekty jsou od pozorovatele blíže a některé dále. Navíc uživatel některé objekty vůbec nemusí vidět. Obdobně není potřeba zobrazovat pohybující se objekty v různé vzdálenosti od pozorovatele se stejnou frekvencí. Pohybující se nebo měnící se objekty, jež jsou od pozorovatele ve větší vzdálenosti, je možné zobrazovat s menší obnovovací frekvencí. Specifikace neřeší, jak často posílat změny objektů v závislosti na jejich vzdálenosti od pozorovatele, ale poskytuje mechanismus, jak dosáhnout co nejlepších výsledků.

Při omezené šířce přenosového pásma může odesílatel vždy přenést jenom omezené množství dat. Je-li k serveru připojeno více klientů, tak mohou dohromady vygenerovat poměrně velký síťový provoz. Kritický je potom

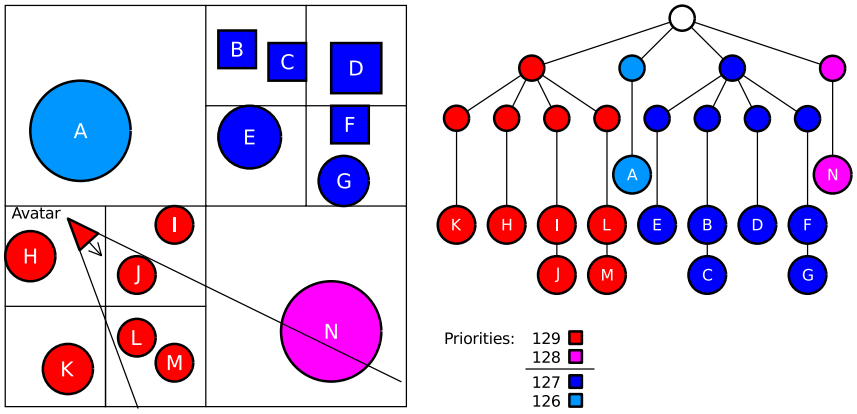
v tomto ohledu přenos dat ze serveru na klienty. Klient přihlášený k velkému počtu uzlů by měl mít možnost určit, jaké uzly pro něj mají prioritu a které by měl server upřednostňovat při přenosu na dané lince. Priorita uzlů má úzkou souvislost s řazením jednotlivých příkazů do front, jež bude popsáno na straně 97.

Struktura příkazu *Node Priority* je uvedena na obrázku 5.15. Priorita uzlu může nabývat hodnot v rozsahu $\langle 0, 255 \rangle$, kde 0 značí nejnižší prioritu a hodnota 255 nejvyšší prioritu. Obecně lze říci, že čím větší má uzel prioritu, tím větší šanci na přenesení mají data v daném uzlu.



Obrázek 5.15: Struktura příkazu pro nastavení priority uzlu

Při odeslání příkazu *Node Priority* si tuto prioritu musí u sebe nastavit i klient. Priority uzlů se nastavují ve stromové struktuře rekurzivně. Je důležité si uvědomit, že priority uzlů jsou na každém spojení různé a klient musí mít možnost nastavit si prioritu i pro uzly, které může pouze číst.



Obrázek 5.16: Schéma možného uspořádání objektů do oktanového stromu

Rychlý pohyb Avatara rozsáhlou 3D scénou může vézt k tomu, že klient bude mít potřebu často měnit prioritu u většiny uzlů. Požadavky na změnu priorit uzlů by potom mohly spotřebovat velkou část šířky pásma.

Z toho důvodu je vhodné omezit množství odesílaných příkazů *Node priority* tak, aby využívaly jenom určitou část šířky pásma. Dále je vhodné objekty ukládat do listů binárního nebo oktanového stromu, aby se mohlo efektivně využívat rekurzivního nastavování priorit uzlů. Zjednodušený příklad uložení objektů do oktanového stromu je uveden na obrázku 5.16. V tomto zjednodušeném příkladu byly priority nastaveny pouze u uzlů, jež jsou potomky kořenového uzlu. Z příkladu je navíc patrné, že není nutné přenášet ke klientovi objekty nacházející se mimo zorný úhel avatara. Uzlům, které jsou například mimo zorný úhel nebo jsou od avatara příliš daleko, by měl klient přiřazovat priorita menší jak 128. Informace o těchto uzlech jsou totiž odeslány teprve, když jsou odeslány informace pro uzly s prioritou vyšší jak 127. Další informace týkající se plánování příkazů na základě priorit jsou uvedeny v kapitole 6.3 na straně 97.

Výchozí hodnota priority nově vytvořených uzlů by měla být nastavena na hodnotu 127.

Dalšího vylepšení při optimálním stanovení priorit uzlů může být dosaženo za použití Kalmanova filtru, jak bylo popsáno v [23]. Pro stanovení priorit v MMORPG lze použít algoritmy popsané v [7].

5.2.12 Tagy a skupiny tagů

Tagy a skupiny tagů jsou přítomny i v původní verzi protokolu. Umožňují do uzlů ukládat jednak dodatečné informace jako například textové poznámky, skutečné jméno autora, datum poslední změny, apod. Tagy a skupiny tagů zároveň umožňují do uzlů ukládat i další užitečné informace jako je hmotnost objektu, teplota, atd.

Koncept nového datového modelu počítá s tím, že spousta specializovaných typů objektů je nahrazena jedním obecným typem uzlů. Význam jednotlivých uzlů by měl být popsán právě pomocí tagů. Zároveň byly tagy uzpůsobeny k efektivnímu ukládání užitečných informací. Často měnit informace v tagech je nyní stejně efektivní jako měnit je ve vrstvách.

Každý objekt může tedy obsahovat tagy, které mají (v rámci své skupiny) své jedinečné jméno, jedinečný číselný identifikátor, typ a hodnotu. Tagy navíc musí být uspořádány do skupin tagů jež mají (v rámci objektu) své jedinečné jméno a identifikátor.

Tag může mít vždy jeden z následujících typů:

- **int8** (signed char)
- **uint8** (unsigned char)
- **int16** (signed short)

- **uint16** (unsigned short)
- **int32** (signed int)
- **uint32** (unsigned int)
- **real32** (float)
- **real64** (double)
- **string8** (unsigned char, *char)

Jména tagů i skupiny tagů mohou být dlouhá pouze 255 znaků. Příklad struktury vytvořené ze skupin tagů a vlastních tagů může mít následující podobu:

- TagGroup(*name:* **Asset**, *ID:* **0**)
  - Tag(*name:* **Name**, *ID:* **0**, *type:* **string8**, *value:* **Obj.001**)
  - Tag(*name:* **Author**, *ID:* **1**, *type:* **string8**, *value:* **Jan Nový**)
  - Tag(*name:* **Version**, *ID:* **2**, *type:* **uint8**, *value:* **5**)
  - Tag(*name:* **Subversion**, *ID:* **3**, *type:* **uint8**, *value:* **3**)
- TagGroup(*name:* **Transformation**, *ID:* **1**)
  - Tag(*name:* **PosX**, *ID:* **0**, *type:* **real32**, *value:* **10.23**)
  - Tag(*name:* **PosY**, *ID:* **1**, *type:* **real32**, *value:* **7.68**)
  - Tag(*name:* **PosZ**, *ID:* **2**, *type:* **real32**, *value:* **-0.03**)
  - Tag(*name:* **QuatW**, *ID:* **3**, *type:* **real32**, *value:* **1.77**)
  - Tag(*name:* **QuatX**, *ID:* **4**, *type:* **real32**, *value:* **-3.2**)
  - Tag(*name:* **QuatY**, *ID:* **5**, *type:* **real32**, *value:* **-7.04**)
  - Tag(*name:* **QuatZ**, *ID:* **6**, *type:* **real32**, *value:* **0.0**)
  - Tag(*name:* **ScaleX**, *ID:* **7**, *type:* **real32**, *value:* **1.0**)
  - Tag(*name:* **ScaleY**, *ID:* **8**, *type:* **real32**, *value:* **2.0**)
  - Tag(*name:* **ScaleZ**, *ID:* **9**, *type:* **real32**, *value:* **1.0**)

Skupiny tagů i samotné tagy obsahují kromě jedinečného jména i jedinečný číselný identifikátor z toho důvodu, že se následně zjednodušují příkazy odkazující se na dané skupiny tagů a tagy. Řetězec by ve většině případů v příkazu zabíral více místa jak dvou bytové ID.

Skupiny tagů

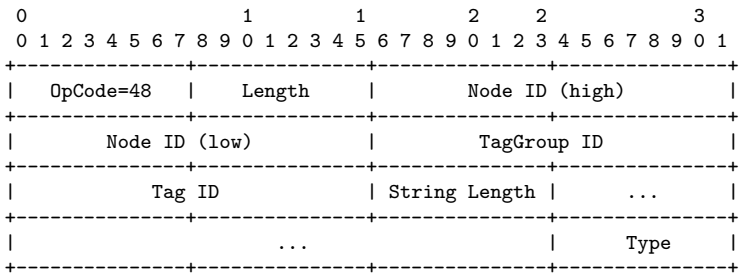
Skupinu tagů lze vytvořit pomocí příkazu *TagGroup Create* jehož struktura je uvedena v příloze na obrázku 10.13 na straně 153. Součástí tohoto příkazu je v rámci objektu jedinečný číselný identifikátor skupiny a jméno skupiny. Příkazy *TagGroup Create* by měl server poslat klientovi pro všechny skupiny tagů v objektu, když se k tomuto objektu přihlásí pomocí příkazu *Node Subscribe*.

Skupinu tagů včetně všech tagů uvnitř této skupiny lze zrušit pomocí příkazu *TagGroup Destroy* jež je uveden v příloze na obrázku 10.14 na straně 153. Skupinu tagů může zrušit každý uživatel, který má právo do daného uzlu zapisovat.

Jména skupin a jejich význam by měly být definovaný v DED. Klient se potom může na základě jména skupiny tagů rozhodnout, jestli obsahu dané skupiny rozumí, případně jestli ho daná skupina zajímá. V takovém případě se může do skupiny tagů přihlásit příkazem *TagGroup Subscribe* jehož struktura je uvedena v příloze na obrázku 10.15 na straně 153. Od skupiny tagů se lze odhlásit pomocí příkazu *TagGroup Unsubscribe*, jehož struktura je uvedena na obrázku 10.16.

Tagy

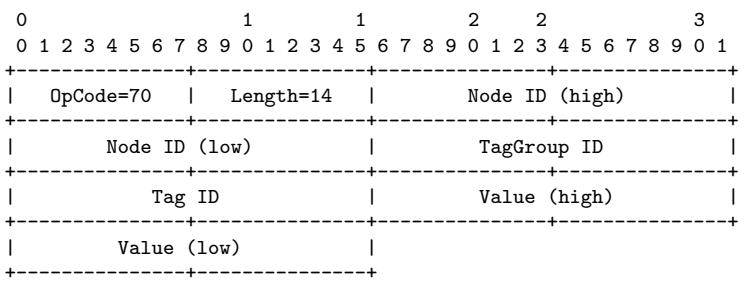
Když se klient přihlásí k neprázdné skupině tagů, server mu pošle sadu příkazů *Tag Create* pro všechny tagy v dané skupině. Struktura daného příkazu je uvedena na obrázku 5.17. Aby se klient nemusel přihlašovat ke každému tagu zvlášť, tak je klient automaticky přihlášen ke všem tagům v dané skupině. Server může poslat vlastní hodnotu tagu v příkazu *Tag Set* teprve, až obdrží od klienta potvrzení o doručení paketu s příkazem *Tag Create*.



Obrázek 5.17: Struktura příkazu pro vytvoření nového tagu

Tagy může pomocí příkazu *Tag Create* vytvářet i klient, ale položka *Tag ID* musí být nastavena na hodnotu 0xFFFF, protože jedinečný identifikátor

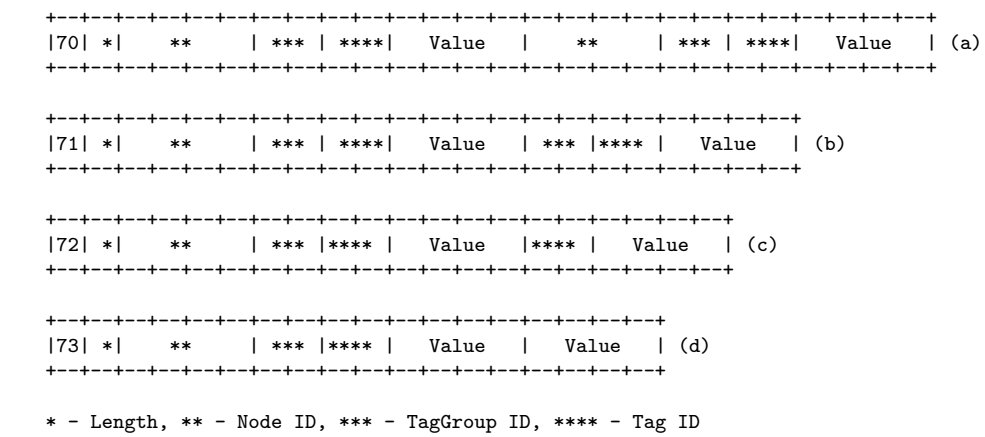
pro tag vybírá opět server. Klient musí příkaz *Tag Create* posílat s jedinečným jménem. Když server obdrží příkaz *Tag Create* s již používaným jménem, tak tento příkaz zahodí, případně může klientovi poslat příkaz s negativním potvrzením dané zprávy.



Obrázek 5.18: Struktura příkazu pro nastavení hodnoty tagu real32

Vlastní hodnota tagu je přenášena v samostatném příkazu *Tag Set*, jenž se liší podle typu tagu a způsobu opakování adresy příkazu. Příkaz pro nastavení tagu *real32* je uveden na obrázku 5.18.

Když se v jednom paketu vyskytuje mnoho příkazů *Tag Set*, které mají totožnou značnou část adresy, tak by bylo neefektivní tuto adresu opakovat. Z tohoto důvodu existují pro každý typ tagu 4 různé varianty příkazu *Tag Set*, jež se liší ve způsobu opakování jednotlivých položek adresy. Celkem je specifikováno 36 variant příkazů *Tag Set*. Způsoby opakování pro typ *real32* jsou uvedeny na obrázku 5.19.



Obrázek 5.19: Jednotlivé varianty opakování adresy příkazu Tag Set (real32)

První varianta (a) opakování je totožná se způsobem opakováním u předchozích příkazů. To znamená, že za položkami *OpCode* a *Length* se opa-

kuje vždy celá adresa příkazu (*Node ID*, *TagGroup ID* a *Tag ID*), která je následována vlastní hodnotou *Value*. Druhá varianta (b) umožňuje uvést položku *Node ID* pouze jednou a u zbylých adres uvádět pouze *TagGroup ID* a *Tag ID*. Třetí varianta (c) umožňuje sdílet i položku *TagGroup ID*. Poslední variantu (d) je možné použít v případě, že odesílatel chce odeslat několik příkazů *Tag Set*, jež mají stejný typ, stejné *Node ID*, *TagGroup ID* a *Tag ID* je u každého následujícího příkazu inkrementováno o jedna.

Každá varianta je vhodná pro jiné kombinace příkazů a odesílatel by měl vždy vybrat vhodnou metodu opakování podle toho, jaké má příkazy v odesílací frontě. Když budeme uvažovat příklad ze strany 87, tak při změně polohy, rotace či velikosti objektu se nejlépe hodí poslední varianta (d) z obrázku 5.19. Změna pozice z příkladu na straně 87 lze pak efektivně přenést pomocí příkazu, jenž je uvedený na obrázku 5.20.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|73| *|    **     | *** |**** |   10.23   |    7.68   |   -0.03   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

* - Length=22, ** - Node ID=65560, *** - TagGroup ID=1, **** - Tag ID=0
```

Obrázek 5.20: Příklad komprese příkazu Tag Set, jež byl použitý pro přenos pozice

5.2.13 Vrstvy

Vrstvy v novém protokolu umožňují sdílet libovolná nestrukturovaná data. Jejich příkazy jsou navrženy pro sdílení objemných dat, jako jsou například souřadnice vertexů, vrcholy plošek, váhy vertexů, apod. Struktura základních příkazů pro vytvoření, zrušení, přihlášení a odhlášení se od vrstvy je uvedena v příloze na straně 159. Když se klient přihlásí k uzlu a ten obsahuje nějaké vrstvy, tak obdrží od serveru sadu příkazů *Layer Create* obsahující identifikátory a typy jednotlivých vrstev. Význam jednotlivých vrstev může být dán například ve skupině tagů jak je uvedeno na následujícím příkladu:

- TagGroup(*name:* **Vertexes**, *ID:* **0**)
  - Tag(*name:* **LayerID**, *ID:* **0**, *type:* **uint16**, *value:* **0**)
  - Tag(*name:* **Components**, *ID:* **1**, *type:* **uint8**, *value:* **3**)

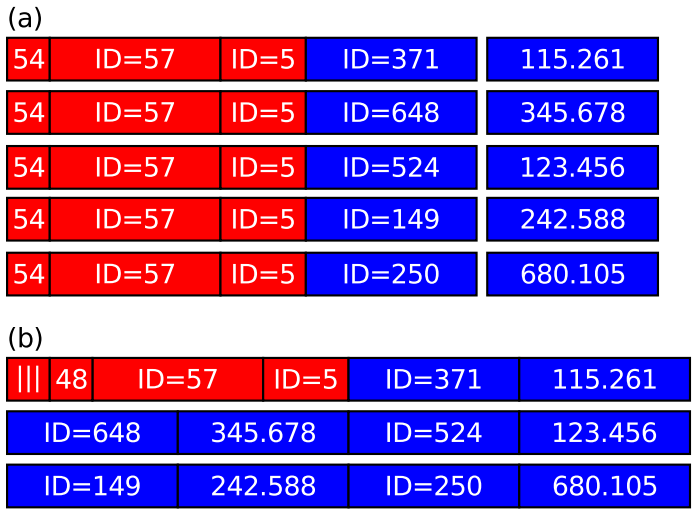
Skupina tagů se jmenuje v tomto příkladu "Vertexes", což značí, že tato skupina bude obsahovat informace o vertexech. Tag s *Tag ID=0* se jménem "LayerID" má hodnotu 0, takže vertexy by měly být obsaženy v nulté vrstvě.

Tag s *Tag ID=1* je pojmenovaný "Components" a jeho hodnota je 3, tudíž každý vertex má vždy tři složky.

Vrstvy mohou obsahovat vždy pouze hodnoty jednoho typu. Podporované typy jsou uvedeny na následujícím seznamu:

- **int8** (signed char)
- **uint8** (unsigned char)
- **int16** (signed short)
- **uint16** (unsigned short)
- **int32** (signed int)
- **uint32** (unsigned int)
- **real32** (float)
- **real64** (double)

Když se klient přihlásí k vybrané vrstvě, tak mu server začne posílat příslušné příkazy *Item Create*, kterých je definováno celkem 32. Opět se liší podle typu přenášených dat a způsobu opakování jednotlivých položek podobně jak to bylo uvedeno na obrázku 5.19.



Obrázek 5.21: Porovnání efektivity využití místa v původním (a) a novém protokolu (b)

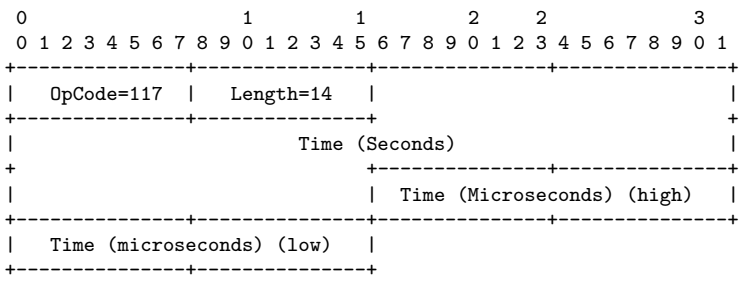
Vrstvy mohou být použity například pro sdílení váhy vertexů. Váhu vertexů může uživatel v grafických aplikacích ovlivňovat pomocí interaktivních nástrojů jako je například štětec. Při kreslení na váhu vertexů Verse klient generuje velké množství příkazů měnící váhu vertexů. Na obrázku 5.21 je potom vidět porovnání využití místa v paketu pří použití původního a nového protokolu. V tomto příkladu příkazy původního protokolu zabírají 75 B a nového protokolu pouze 48 B.

5.3 Časové značky

V předchozích dvou částech byly uvedeny metody pro sdílení dat. Klient může například pomocí vrstvy sdílet na serveru nejen polohu objektu $s = (x, y, z)$, ale může to být i třeba jeho rychlost $\vec{v} = (v_x, v_y, v_z)$ a zrychlení $\vec{a} = (a_x, a_y, a_z)$. Důvodem pro sdílení rychlosti a zrychlení může být snaha dosáhnout spojitého pohybu objektů i při ztrátě některých paketů. Klient přijímající tyto informace ovšem potřebuje ke správnému dopočítání aktuální polohy objektu pomocí vztahu 5.3 i informaci o času t_0, kdy byla poloha, rychlost a zrychlení u daného objektu stanoveny.

$$\begin{aligned} x(t_1) &= x(t_0) + v_{x(t_0)}(t_1 - t_0) + \tfrac{1}{2}a_{x(t_0)}(t_1 - t_0)^2 \\ y(t_1) &= y(t_0) + v_{y(t_0)}(t_1 - t_0) + \tfrac{1}{2}a_{y(t_0)}(t_1 - t_0)^2 \\ z(t_1) &= z(t_0) + v_{z(t_0)}(t_1 - t_0) + \tfrac{1}{2}a_{z(t_0)}(t_1 - t_0)^2 \end{aligned} \tag{5.1}$$

Klient posílající informaci o rychlosti a zrychlení musí tedy posílat i informaci o času t_0. Jelikož se simulace pohybu více objektů většinou vypočítává pro jeden časový okamžik, tak je efektivní hodnotu t_0 sdílet pro více příkazů pomocí jednoho příkazu *Time Stamp* jež je uveden na obrázku. Tento čas je potom platný pro všechny následující příkazy v daném paketu.



Obrázek 5.22: Struktura příkazu pro nastavení časové značky

Chceme-li vyjádřit, že v daném paketu jsou následující pakety bez časové

značky, tak je potřeba vložit do paketu příkaz *Time Stamp*, kdy položka *Time (seconds)* má hodnotu 2^{64} a položka *Time (Microseconds)* má hodnotu 2^{32}.

Čas je vyjádřen pomocí dvou hodnot podobně jako UNIXový čas. Hodnota (64-bitový integer) v sekundách vyjadřuje UTC čas od půlnoci 1. ledna 1970. Druhá hodnota vyjadřuje čas v mikrosekundách od poslední ukončené sekundy.

Další podmínkou pro korektní fungování časových značek je synchronizace času na jednotlivých klientech. Protokol Verse žádný takový mechanismus neposkytuje, takže klienti používající příkaz *Time Stamp* si musí synchronizovat čas například pomocí protokolu NTP [27]. NTP protokol umožňuje v závislosti na kvalitě spojení nastavit čas s přesností až 0,2 ms.

Pokud nastane situace, že by klient neměl čas synchronizovaný korektně a hodiny by se mi zpožďovaly, tak se může stát, že čas t_0přijatý pomocí příkazu *Time Stamp* bude napřed před jeho aktuálním časem t_1. Výpočet polohy pomocí vztahu 5.3 by potom vedl k nekorektním výsledkům. V případě, že je $t_0 > t_1$ není možné provádět estimaci aktuální polohy objektu, ale je nutné použít pouze přijatou informaci o poloze a rychlost i zrychlení ignorovat.

Kapitola 6

Implementace

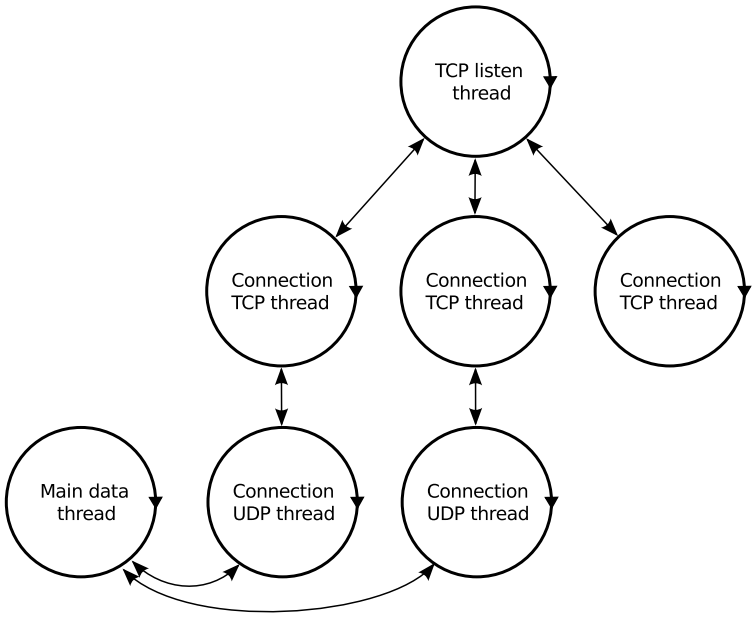
Tato kapitola neobsahuje pouze popis implementace, ale měla by dát návrh a doporučení, jak efektivně provádět implementaci Verse serveru i klienta. Jedná se především o implementaci resend mechanismu, kde je vyžadováno, aby byla posílána pouze aktuální data.

Výsledná implementace protokolu Verse je naprogramována v jazyku C a využívá z knihovny *OpenSSL* implementaci protokolu TLS a DTLS. Dále je využívána knihovna *pthread* pro práci s vlákny. Použité technologie umožňují snadné portování na různé platformy.

6.1 Server

Při implementaci Verse serveru byl kladen důraz na dosažení maximálního výkonu a propustnosti. Proto byl implementován jako vícevláknová aplikace. Server poslouchá na požadavky klientů v samostatném vlákně pomocí TCP soketu, který je vytvořen pomocí systémového volání *listen()*. Klientův požadavek na nové spojení je detekován pomocí systémového volání *select()*. Server následně vytvoří pro nové spojení samostatné vlákno. Teprve v novém vlákně se provede TCP a TLS handshake. Po autentizaci uživatele a dohodnutí nového UDP spojení je pro UDP spojení vytvořeno další tzv. datagramové vlákno. Zde může proběhnout DTLS handshake a následně handshake datagramového spojení. Vlastní komunikace s klientem probíhá v tomto vlákně.

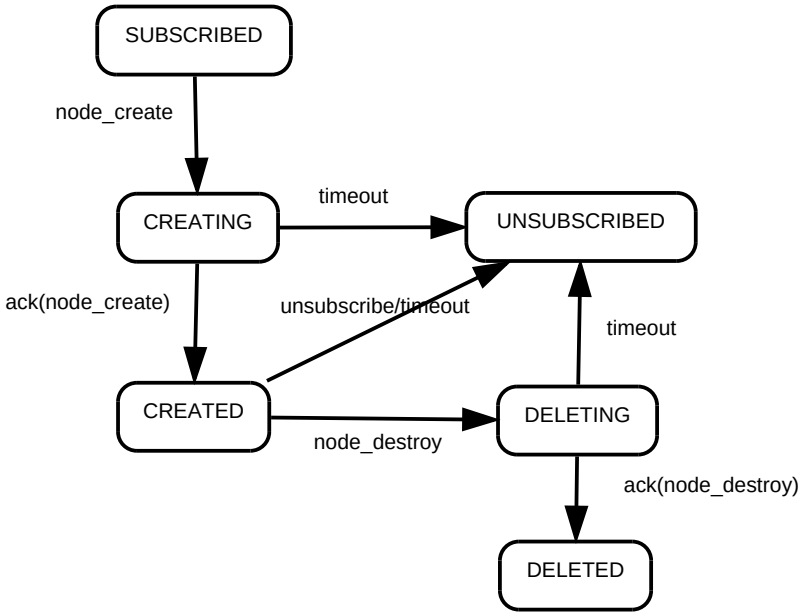
Když se některý z handshaků nepovede nebo je dané spojení zrušeno, tak jsou obě vlákna ukončena a příslušné systémové prostředky jsou uvolněny pro další spojení. Kromě výše zmíněných vláken je při startu systému vytvořeno jedno datové vlákno, které spravuje sdílená data. Komunikace mezi datovým vláknem a datagramovým vláknem se provádí pomocí speciálních front, které



Obrázek 6.1: Vlákna Verse serveru

budou popsány v části 6.3 na straně 97. Datové vlákno při přijetí zprávy z fronty zpráv nejprve zkontroluje korektnost přijaté zprávy (platnost ID, přístupová práva, atd.). Když přijatá zpráva neobsahuje žádnou chybu, tak pomocí ní nastaví vlastní kopii sdílených dat a následně tuto zprávu zařadí do fronty zpráv klientů, kteří byli přihlášeni k uzlům, skupinám tagů nebo vrstvám. Datové vlákno by zároveň mělo zajistit, aby klient neobdržel příkaz rušící nějakou entitu (uzel, skupinu tagů, atd.), aniž by před tím obdržel odpovídající příkaz pro vytvoření dané entity. Každá výše zmíněná entita by si měla uchovávat pro každého přihlášeného klienta záznam o tom, v jakém stavu se z jeho pohledu nachází. Příslušný stavový diagram je uveden na obrázku 6.2.

Obecně platí doporučení, aby každé spojení mělo minimálně jedno vlákno a jeden samostatný TCP soket a UDP soket. Implementace vlastního vlákna pro každé spojení umožňuje garantovat jednotlivým spojením férové zacházení. Požadavek na vlastní UDP soket pro každé spojení vychází ze skutečnosti, že většina operačních systémů alokuje pro vstupní buffer UDP soketu přibližně 100 KB. Z tohoto důvodu je nezbytné, aby se vlákno snažilo co nejrychleji tento buffer vyprazdňovat pomocí systémového volání *recf()*. Velikost vstupního bufferu pro všechny UDP sokety lze sice zvětšit netriviálním zásahem administrátora operačního systému, ale má to samozřejmě svá bezpeč-



Obrázek 6.2: Stavy uzlu

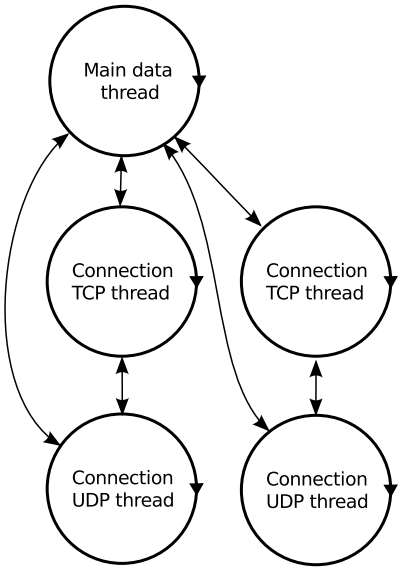
nostní rizika.

Původní Verse server byl implementován jako jednovláknová aplikace, která pro všechna spojení používala jeden soket. Server potom při velkém počtu klientů a velkém datovém přenosu nestíhal vybírat dostatečně rychle vstupní buffer UDP soketu a operační systém značnou část přenesených paketů zahazoval.

6.2 Klient

Pro tvorbu Verse klientů byla vytvořena knihovna a navrženo nové API, jehož podstatná část již je plně implementována. Komunikace klienta s každým serverem probíhá také pomocí dvou samostatných vláken, která jsou vytvořena po zavolání funkce *verse_send_connect_request()*. Datagramové vlákno je vytvořeno opět až po úspěšné autentizaci a dohodnutí nového datagramového spojení. Samotná výměna dat mezi hlavním vláknem a datagramovým vláknem probíhá pomocí funkcí začínajících řetězcem *verse_send_*, jež jsou uvedeny v příloze na straně 165.

Aby bylo hlavní vlákno schopné dostávat data od datagramového vlákna, tak si musí klient zaregistrovat příslušné callback funkce, jež jsou opět uvedeny v příloze na straně 165. Vlastní výměna dat mezi vlákny se provádí opět pomocí speciální fronty příkazů. Výhoda více vláken u klienta spočívá



Obrázek 6.3: Vlákna verse klienta

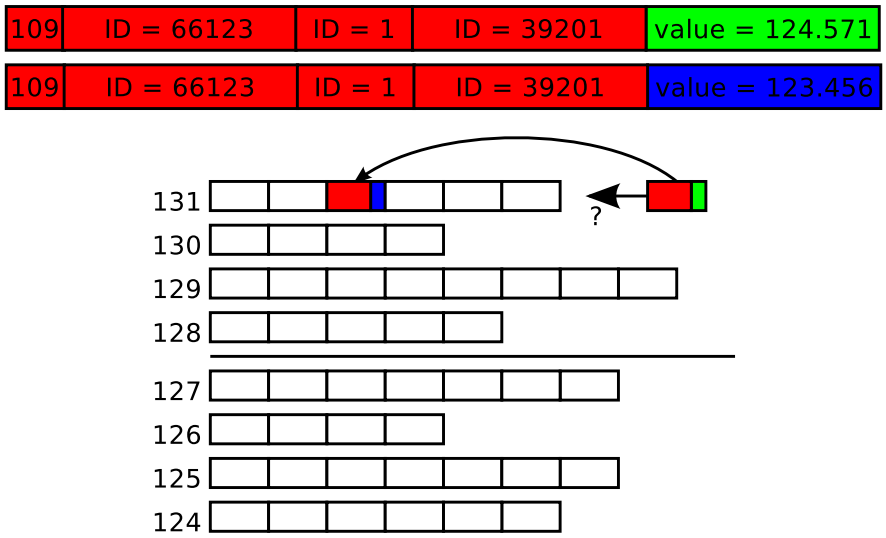
v tom, že hlavní vlákno může být zaneprázdněno například renderováním rastrového obrázku, výpočtem fyzikální simulace, atd. Datagramové vlákno mezitím provádí komunikaci se serverem a uchovává přijatá data dokud si o ně hlavní vlákno neřekne zavoláním funkce *verse_callback_update()*. Verse klient samozřejmě může komunikovat s více servery zároveň a pro každé spojení jsou vytvořena dvě vlákna, jak je uvedeno na obrázku 6.3.

Původní implementace protokolu Verse také poskytovala knihovnu umožňující implementaci Verse klientů. Komunikace ovšem neprobíhala v samostatném vlákně, ale jen v okamžicích, kdy Verse klient zavolal buď nějakou funkci začínající řetězcem *verse_send_* nebo funkci *verse_callback_update()*. Když bylo hlavní vlákno klienta zaneprázdněno například renderováním, které trvalo déle jak 30 sekund, server komunikaci s klientem ukončil. Programátor se mohl pokusit řešit tento nedostatek implementací komunikace ve vlastním vlákně, ale knihovna implementující původní protokol Verse nebyla naprogramována jako vláknově bezpečná. Například výsledek volání funkce *verse_session_get()* záviselo na hodnotě globální proměnné.

6.3 Fronta příkazů

Jednotlivá vlákna na serveru i klientovi si mezi sebou potřebují efektivně předávat data. Kdyby byla použita jednoduchá fronta příkazů, mohlo by dojít

k situaci, kdy hlavní vlákno zaplňuje frontu příkazů rychleji než je datagramové vlákno schopné odesílat. Následkem toho by se mohly v jednom paketu ocitnout dva příkazy se stejnou adresou, což je značně neefektivní, protože první příkaz by byl při přijetí u příjemce okamžitě přepsán druhým příkazem. Z tohoto důvodu byla implementována speciální fronta, jejíž struktura je naznačena na obrázku 6.4.



Obrázek 6.4: Přidání příkazu do fronty příkazů

Fronta pro předávání příkazů se skládá z několika speciálních front a 256 prioritních front, jež odpovídají jednotlivým prioritám uzlů. Frontu pro prioritu p_i; $i \in \langle 0, 255 \rangle$ si označíme jako F_i. Každý uzlový příkaz *cmd* obsahuje svoji adresu $adr = (node\_id, \ldots)$ a vlastní data. Na každém spojení lze pro každý uzel určit jeho prioritu $p_i = f(node\_id)$. Příkaz je následně podle *node_id*, který odkazuje, vložen do odpovídající fronty F_i.

Nový příkaz cmd_n nesmí být automaticky vložen na konec fronty F_i. Pokud totiž fronta již obsahuje příkaz cmd_o se stejnou adresou jako příkaz cmd_n, tak původní příkaz cmd_o musí být nahrazen novým. Vyhledání příkazu se stejnou adresou by se nemělo provádět sekvenčním prohledáváním dané fronty, ale je vhodné implementovat k vyhledávání hašovací tabulku, která na základě adresy nového příkazu cmd_n buď rychle najde existující příkaz cmd_o nebo ohlásí, že příkaz s touto adresou se ve frontě F_i nenachází. V druhém případě je příkaz cmd_n vložen na konec fronty.

K frontě příkazů přistupují vždy dvě vlákna, takže je nutné zajistit, aby s danou frontou F_i pracovalo vždy jen jedno vlákno. Tento požadavek byl

implementováno pomocí mutexů. Každá fronta F_i je zamknutá během operací přidání a odebrání příkazu.

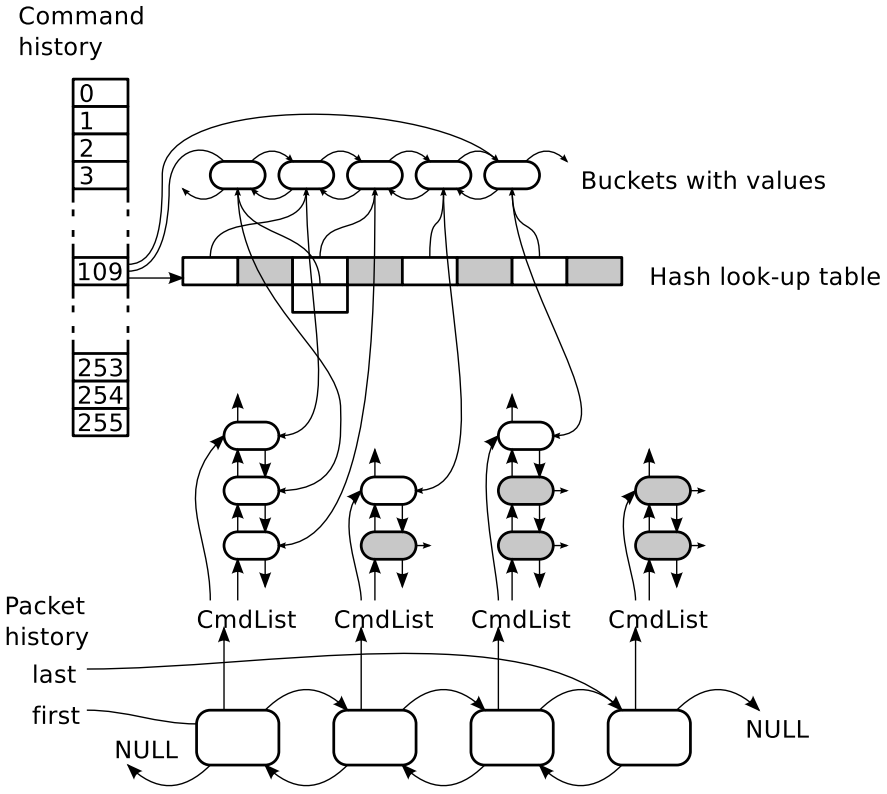
Komunikační vlákno by mělo příkaz vybírat z jednotlivých front F_i následujícím způsobem. Nejprve je nutné vybrat všechny příkazy z jednotlivých front F_i; $i \in \langle 128, 255 \rangle$. Do každého paketu jsou pakety vkládány pomocí algoritmu Wighted Fair Queuing (WFQ), jak je popsáno v [24]. Teprve, když jsou fronty v rozsahu $\langle 128, 255 \rangle$ prázdné, tak je možné začít vyprazdňovat fronty F_i; $\langle 0, 127 \rangle$ opět pomocí WFQ. Navržený způsob může za určitých podmínek způsobit vyhladovění front z rozsahu $\langle 0, 127 \rangle$, což je paradoxně žádoucí chování. Klient by měl uzlům nastavovat prioritu menší jak 128 v případě, že není nutné informace z takových uzlů dostávat, protože jsou například mimo jeho zorný úhel.

Klient i server musí mít nastavenou maximální velikost fronty zpráv pro příchozí spojení a s každým přijatým paketem spočítat dostupné volné místo. Dostupné volné místo se pak přepočítá na hodnotu *rwin* (viz. Flow Control na straně 4.11), která je deklarována v každém odeslaném paketu v položce *Window*.

6.4 Historie příkazů

Resend mechanismus datagramového spojení vyžaduje, aby byly přeposlány pouze aktuální příkazy. Když je v historii odeslaných paketů uložen příkaz cmd_o a dojde k odeslání nového paketu obsahujícího příkaz cmd_n se stejnou adresou, tak při uložení nového paketu musí dojít k zneplatnění starého příkazu cmd_o. Obdrží-li odesílatel příkaz *NAK(node_id)*, tak musí odesílatel z historie odeslaných paketů načíst příslušný paket a všechny platné příkazy cmd_l musí vložit na začátek odesílacích front F_i. Pokud se ve frontě již nachází novější příkaz se stejnou adresou jako má příkaz cmd_l, příkaz cmd_l nebude přeposlán, protože již ve frontě existuje jeho novější varianta.

Způsob implementace historie příkazů je uveden na obrázku 6.5. Nové pakety jsou přidávány na konec dvojcestného dynamického seznamu. Každý paket uložený v historii paketů v obsahuje své ID a dvoucestný dynamický seznam uzlových příkazů, které obsahoval. Samotné příkazy v tomto seznamu ovšem obsahují pouze odkaz na buket s adresou příkazu a jeho daty. Tento buket zároveň obsahuje zpětný odkaz na příkaz v dynamickém seznamu odeslaného paketu. Navržená struktura umožňuje efektivně přidávat odeslané pakety a jejich příkazy a jednoduše zneplatnit zastaralé příkazy.



Obrázek 6.5: Schéma historie odeslaných paketu

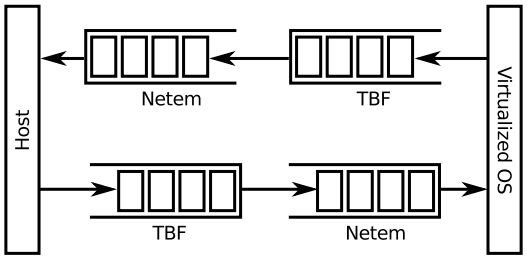
Kapitola 7

Výsledky měření

Pro testování byla vytvořena speciální klient-server aplikace, která simulovala velké množství uživatelů ASVR pomocí částicového systému.

7.1 Podmínky experimentů

Síťové protokoly byly testovány jednak v reálném síťovém prostředí, ale pro jejich vzájemné porovnání bylo potřeba zvolit jiný přístup. Bylo nezbytné nastavit přesné parametry datového okruhu a zajistit, aby tento okruh nepoužívaly další přenosy. Toho bylo dosaženo vytvořením spojení mezi hostujícím a virtualizovaným operačním systémem, kde obě linky byly upraveny pomocí nástroje Traffic Control (tc) upravující traffic control jádra operačního systému Linux. Queueing discipline (qdisc) Netem [16] byla použita k nastavení zpoždění a rozptylu zpoždění. TBF qdisc byla použita k omezení šířky pásma.



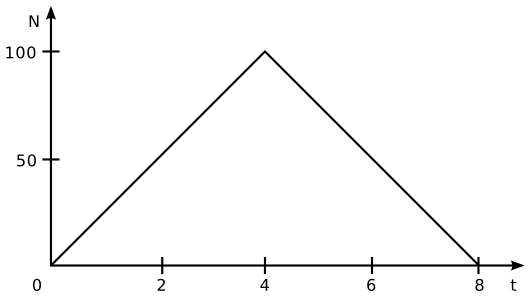
Obrázek 7.1: Schéma omezení datového okruhu

Na obrázku 7.1 je patrné, že bylo nutné Netem qdisc připojit na konec (ingress) a TBF qdisc na začátek (egress) každé linky. Důvod je ten, že obě qdics jsou classless a nebylo možné je na sebe připojit přímo. Je důležité

zmínit, že nemodifikované linky měly průměrné zpoždění okolo 0,5 ms a jejich delay jitter byl 0,03 ms. Nejmenší nastavované hodnoty byly minimálně desetkrát větší.

7.2 Metody experimentů

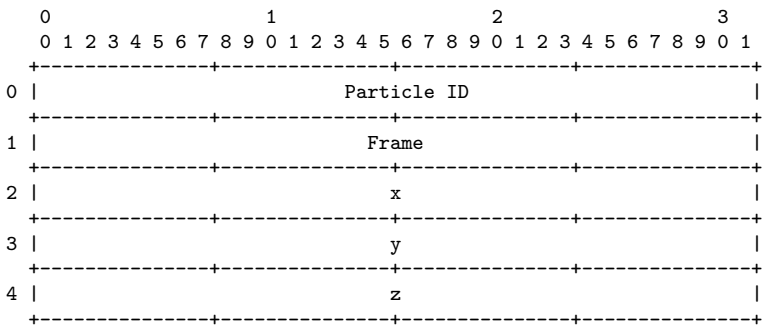
Pro experimenty byly vygenerovány v programu Blender dva částicové systémy. Pro testování na virtuální lince byl vytvořen částicový systém čítající 100 částic a pro testování ve skutečném síťovém byl vygenerovaný částicový systém obsahující 1000 částic. Oba částicové systémy byly vždy uloženy na stranu klienta i serveru. 100 částic umožňovalo vytvářet při testování na virtuální lince názornou vizualizaci a zároveň bylo možné efektivně omezit šířku pásma pro generovaný datový tok.



Obrázek 7.2: Časový průběh počtu posílaných částic

Částicový systém byl vygenerován pro animaci s 25 snímky za vteřinu. Všechny testované transportní protokoly používaly na aplikační vrstvě velmi jednoduchý protokol. Klient zahajuje komunikaci se serverem zasláním jednoduché zprávy. Server na tuto žádost reaguje tím, že začne klientovi posílat každých 40 ms pozice všech pohybujících se částic. Prvních 10 příchozích paketů nebo zpráv se použilo k výpočtu průměrného zpoždění na lince. Pozice každé pohybující se částice byla uložena v jednoduché zprávě, která byla totožná pro všechny testované protokoly. Zpráva měla následující tvar:

Ve zprávě byla přenášena pouze poloha částic, protože částicový systém měl simulovat stochastický pohyb a další činnosti generované uživateli pracujícími v aplikacích sdílené virtuální reality (ASVR). Pro uživatele virtuální reality je velmi rušivý pohyb objektů, který by měl být spojitý, ale z různých důvodů je tento pohyb přerušován. Jestli se pohyb bude jevit uživateli jako nespojitý ovlivňuje mnoho faktorů: rozlišení zobrazovacího zařízení, vzdálenost uživatele od zobrazovacího zařízení, apod. Z tohoto důvodu byl uvažován nejhorší možný případ, kdy ztrátu jediné polohy částice je schopný uživatel



Obrázek 7.3: Struktura zprávy pro posílání částic

zaznamenat. Zpoždění částice větší jak 40 ms od očekávaného zpoždění bylo tudíž vizualizováno jako problematické zpoždění jak je vidět na obrázku 7.5. Když je v některých ASVR kladen velký důraz na plynulost pohybu objektů, tak je vhodné kromě polohy přenášet i čas, rychlost a zrychlení, které umožňují dopočítat pohyb objektu v případě ztráty přenášených dat.

Každý transportní protokol byl otestován pro několik hodnot zpoždění a jeho rozptylu. Rozložení pravděpodobnosti zpoždění paketu se řídilo normálním rozdělením. Použité hodnoty jsou uvedeny v následující tabulce:

	Zpoždění [ms]	Rozptyl zpoždění [ms]
1.	5	2,5
2.	10	3
3.	20	5
4.	40	10
5.	80	20

Tabulka 7.1: Testovaná zpoždění a jejich rozptyl

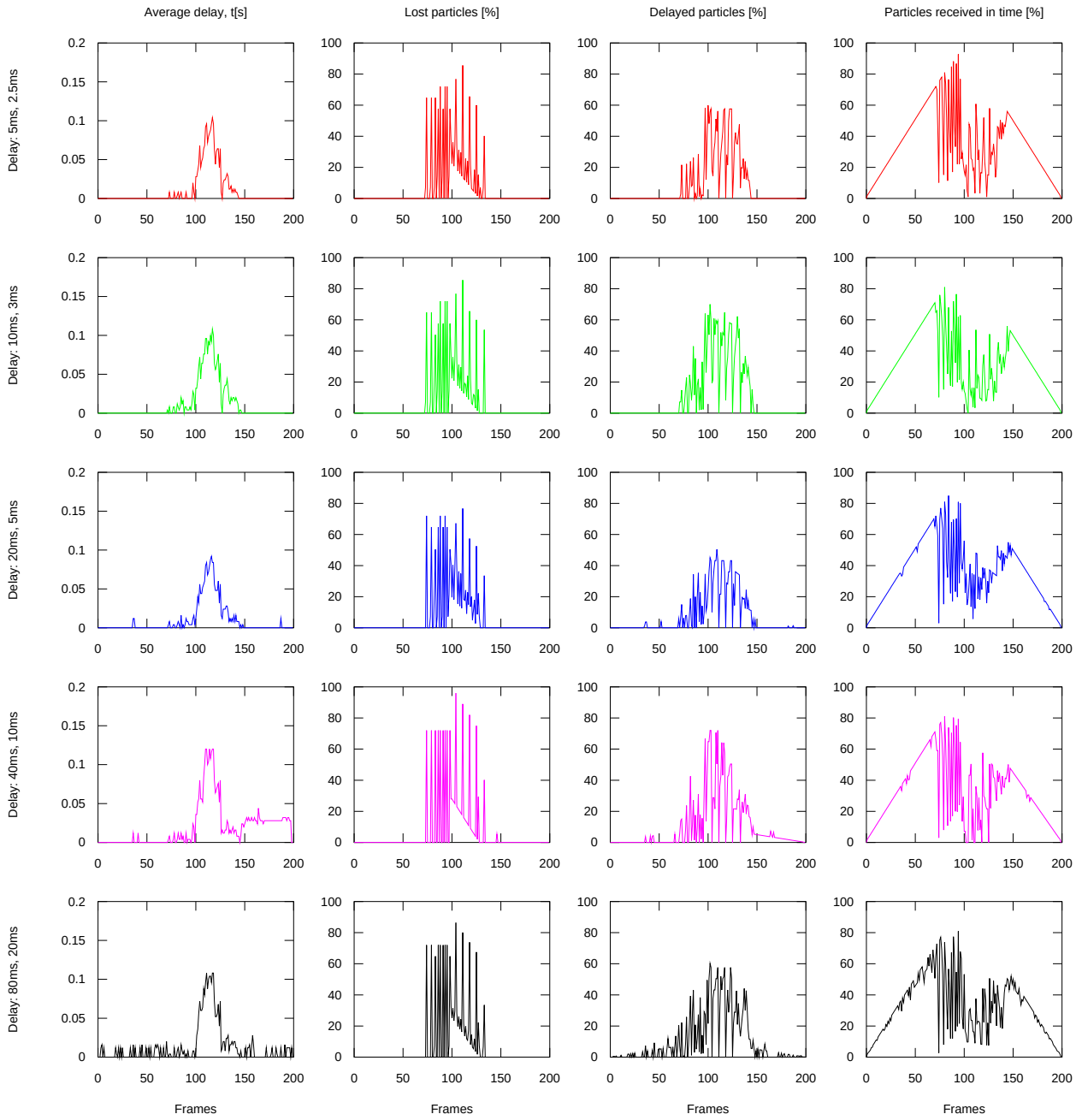
7.3 Transportní protokoly

7.3.1 UDP

User Datagram Protocol (UDP) [29] byl první testovaný protokol. UDP je nespolehlivý datagramový protokol široce používaný v herních aplikacích pro svoji jednoduchost, nízké latence a možnost použít multicast. Obecnou nevýhodou protokolu UDP je fakt, že nemá žádný Congestion Control (CC) mechanismus a jeho použití může způsobit úplné zahlcení přenosových cest (Congestion Collapse). Použití čistého UDP v ASVR je možné pouze tehdy,

když není vyžadována spolehlivost přenášených dat. Výsledky měření protokolu UDP jsou uvedeny na obrázku 7.4 na straně 105.

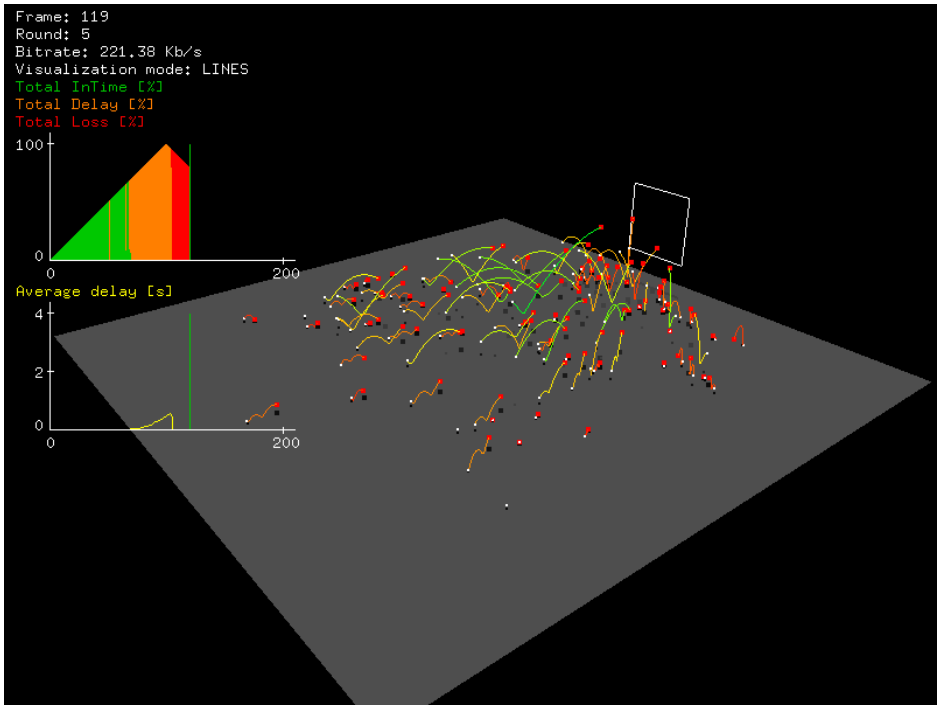
Z výsledků měření je patrné, že všechny sledované parametry (průměrná doba zpoždění, počet zpožděných, ztracených a včas doručených částic) jsou nezávislé na nastaveném zpoždění a rozptylu zpoždění paketů na lince. Ani zvyšující se velikost rozptylu zpoždění paketů na dané lince nezhoršuje výsledky, protože rozptyl zpoždění by musel být výrazně vyšší jak 40 ms ($\tilde{2}5$ FPS), aby výrazně ovlivnil výsledky. Jediným faktorem, který ovlivňoval spojitost pohybu částic, byla šířka přenosového pásma.



Obrázek 7.4: Výsledky měření za použití protokolu UDP

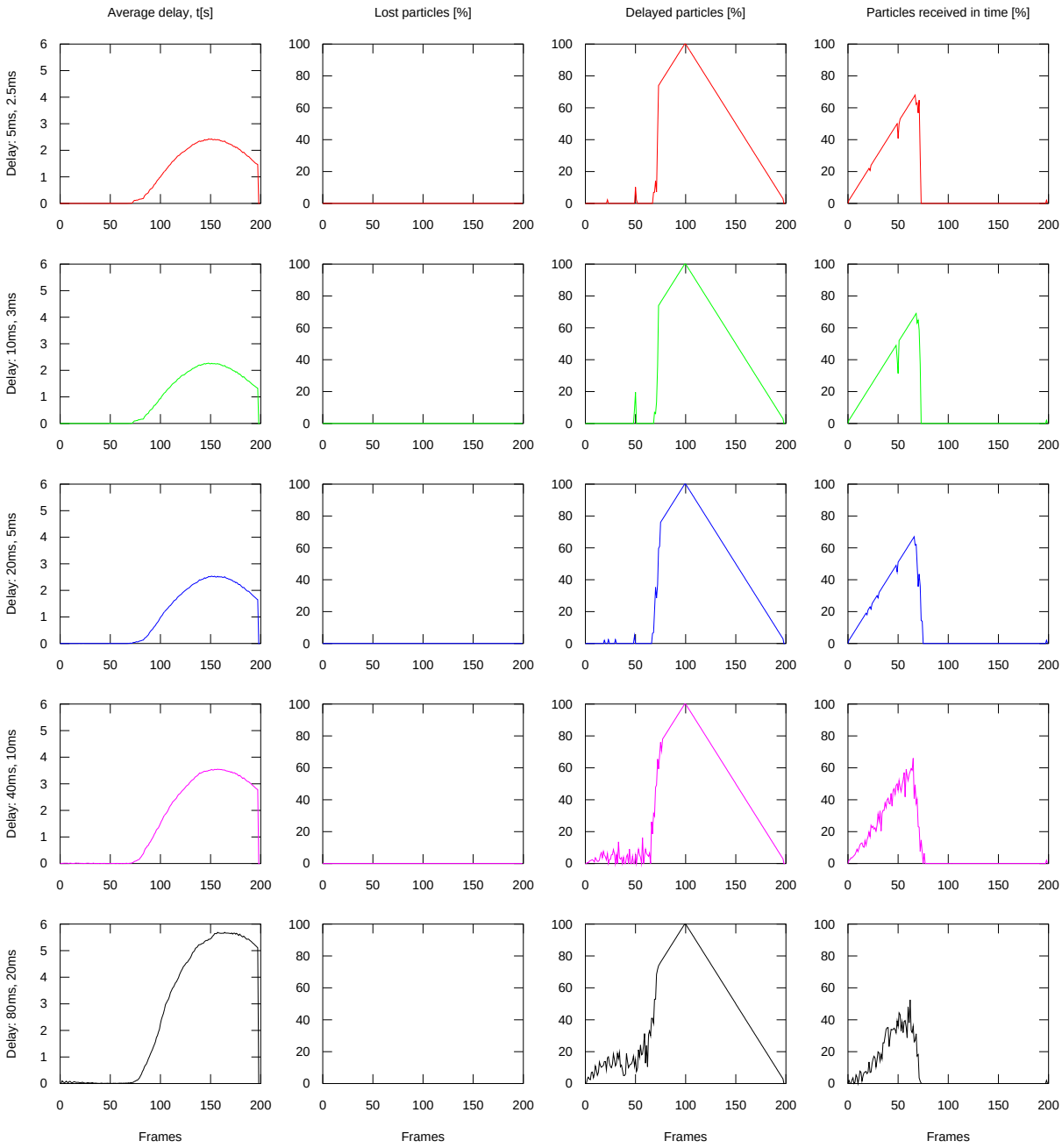
7.3.2 TCP

Transmission Control Protocol (TCP) [30] není běžně používán v ASVR. Výjimkou jsou pouze herní aplikace provozované na mobilních zařízeních připojených do sítě mobilních operátorů. Mobilní operátoři v mnoha případech neumožňují používat jiný protokol než TCP. Obecně je TCP nevhodný pro real-timový přenos dat, protože TCP zajišťuje doručení všech dat ve správném pořadí. Přeposílání ztracených dat vede k velkým zpožděním, která způsobují v ASVR trhaný pohyb objektů. Výsledky měření protokolu TCP jsou uvedeny na obrázku 7.6 na straně 107.



Obrázek 7.5: Screenshot klientské aplikace vizualizující zpoždění částic (protokol TCP)

Výsledky měření potvrdily, že protokol TCP je pro ASVR naprosto nevhodný, protože zpoždění částic se i pro nejnižší nastavené zpoždění paketů dlouho drželo kolem jedné sekundy. Navíc zpoždění částic několikanásobně vzrůstalo pro vyšší zpoždění paketů na lince.



Obrázek 7.6: Výsledky měření za použití protokolu TCP

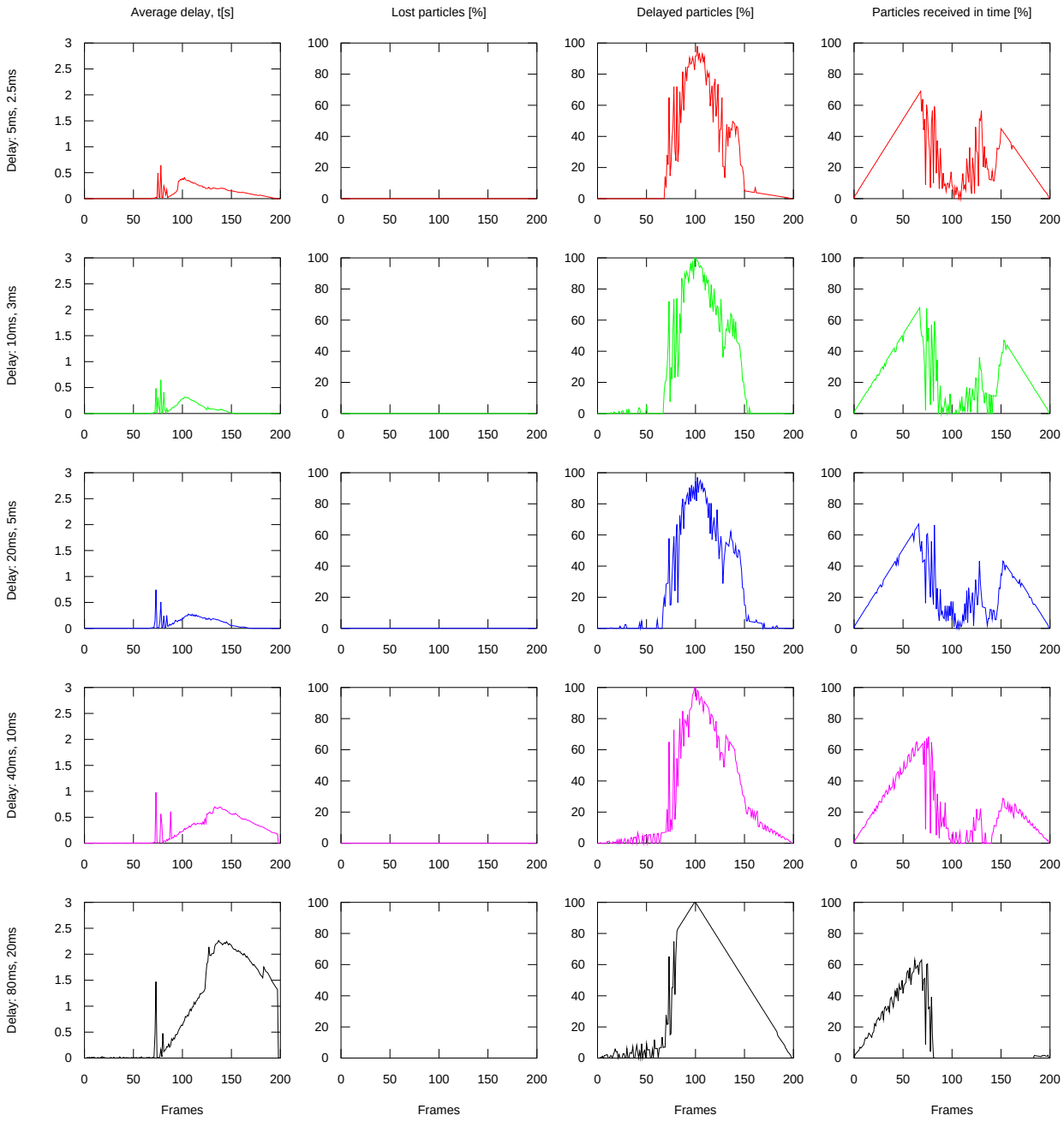
7.3.3 SCTP

Stream Control Transmission Protocol (SCTP) [34] je moderní transportní protokol, který posílá data pomocí několika nezávislých kanálů. SCTP není proudově orientovaný transportní protokol, ale data jsou přenášena ve zprávách. Doručení zpráv ve stejném pořadí v jakém byly zprávy odeslány není zaručeno. Navíc lze u každé zprávy nastavit časový limit po který se má odesílatel snažit zprávu přeposlat. V takovém případě hovoříme o částečně spolehlivé variantě protokolu SCTP, protože daná zpráva nemusí být doručena.

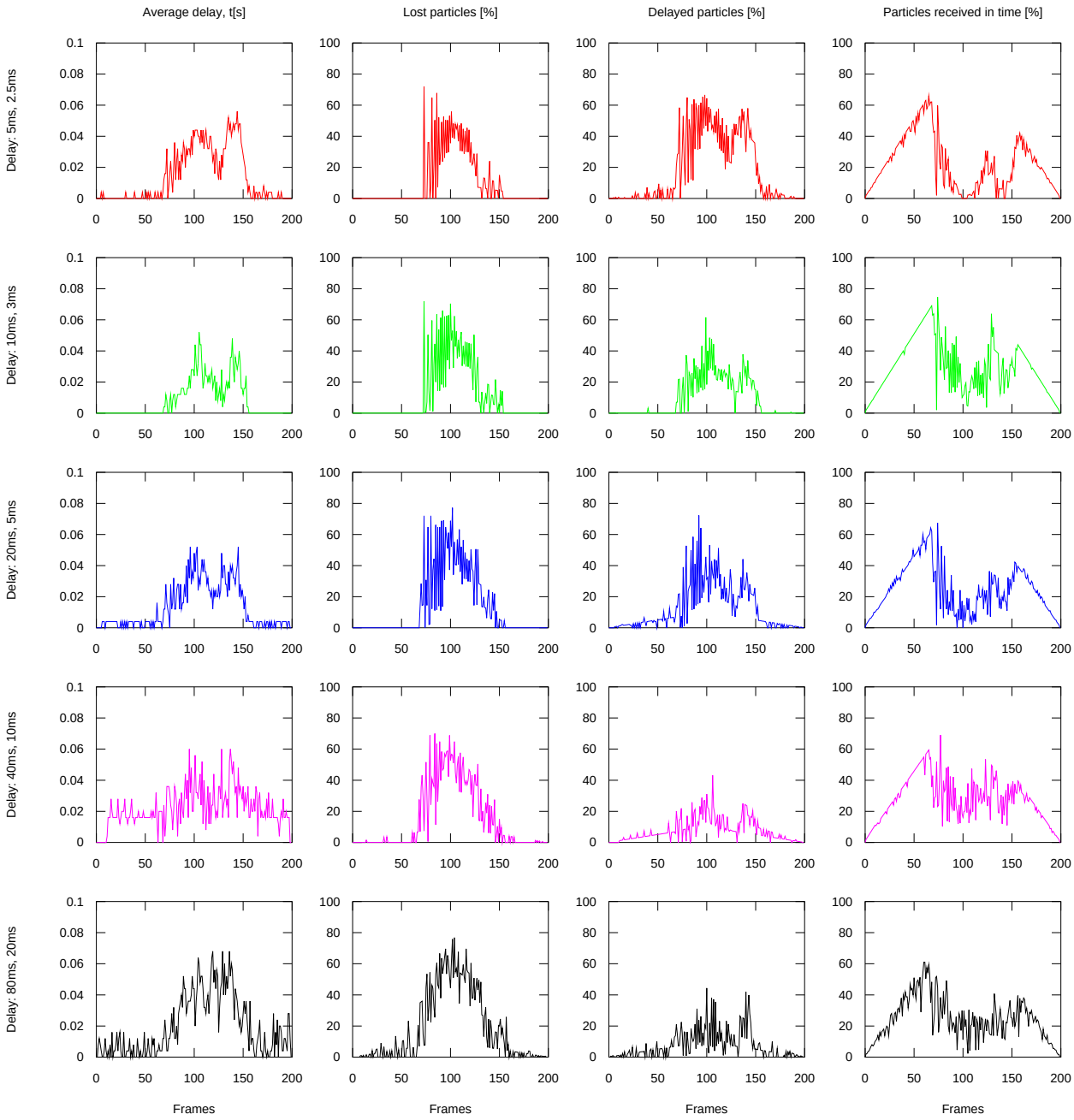
Výsledky měření spolehlivé varianty protokolu SCTP jsou uvedeny na obrázku 7.7 a výsledky pro částečně spolehlivou variantu jsou na obrázku 7.8 na straně 110.

Spolehlivá varianta dává lepší výsledky jak protokol TCP, ale přesto použití SCTP je pro ASVR nevhodné. Spolehlivá varianta se snaží podobně jako TCP přeposlat všechny částice, přestože jejich odeslání již není zapotřebí.

Částečně spolehlivá varianta SCTP dává podobné výsledky jako protokol UDP. Časový limit u všech odeslaných zpráv byl nastaven na polovinu periody mezi dvěma obrazovými snímky, aby měl server případně čas ztracenou zprávu odeslat. Časově omezené přeposílání ztracených zpráv ovšem mělo spíše negativní vliv na celkovou spojitost pohybu částic, protože přeposílání spíše zhoršovalo zahlcení linky. Patrné to je u velkého zpoždění paketů na lince.



Obrázek 7.7: Výsledky měření za použití spolehlivé varianty protokolu SCTP



Obrázek 7.8: Výsledky měření za použití částečně spolehlivé varianty protokolu SCTP

7.3.4 DCCP

Datagram Congestion Control Protocol (DCCP) [25] je moderní datagramový transportní protokol navržený primárně jako transportní protokol pro streamování multimediálnách dat. DCCP implementuje několik variant Congestion Control [12] [13]. Mezi jeho další vlastnosti patří čtyřcestný handshake, přátelské ukončení spojení, dohadování o vlastnostech spojení a v neposlední řadě podporuje Explicit Congestion Notification (ECN). Díky tomu by měl být congestin control efektivnější, jak jeho implementace na aplikační vrstvě jak to provádí například Real-time Transport Protocol (RTP) [14] v kombinaci s RTP Control Protocol (RTCP).

Tato práce bohužel neobsahuje výsledky testování protokolu DCCP, protože při ztrátě paketů docházelo k nekorektnímu chování operačního systému (pád virtualizovaného stroje, alokování velkého množství paměti, apod.). Mnohé internetové zdroje [35] popisují toto chování a v budoucnu snad dojde k jejich nápravě. V současné době se protokol DCCP vzhledem ke stavu implementace nejeví jako použitelný transportní protokol pro ASVR.

7.3.5 Další transportní protokoly

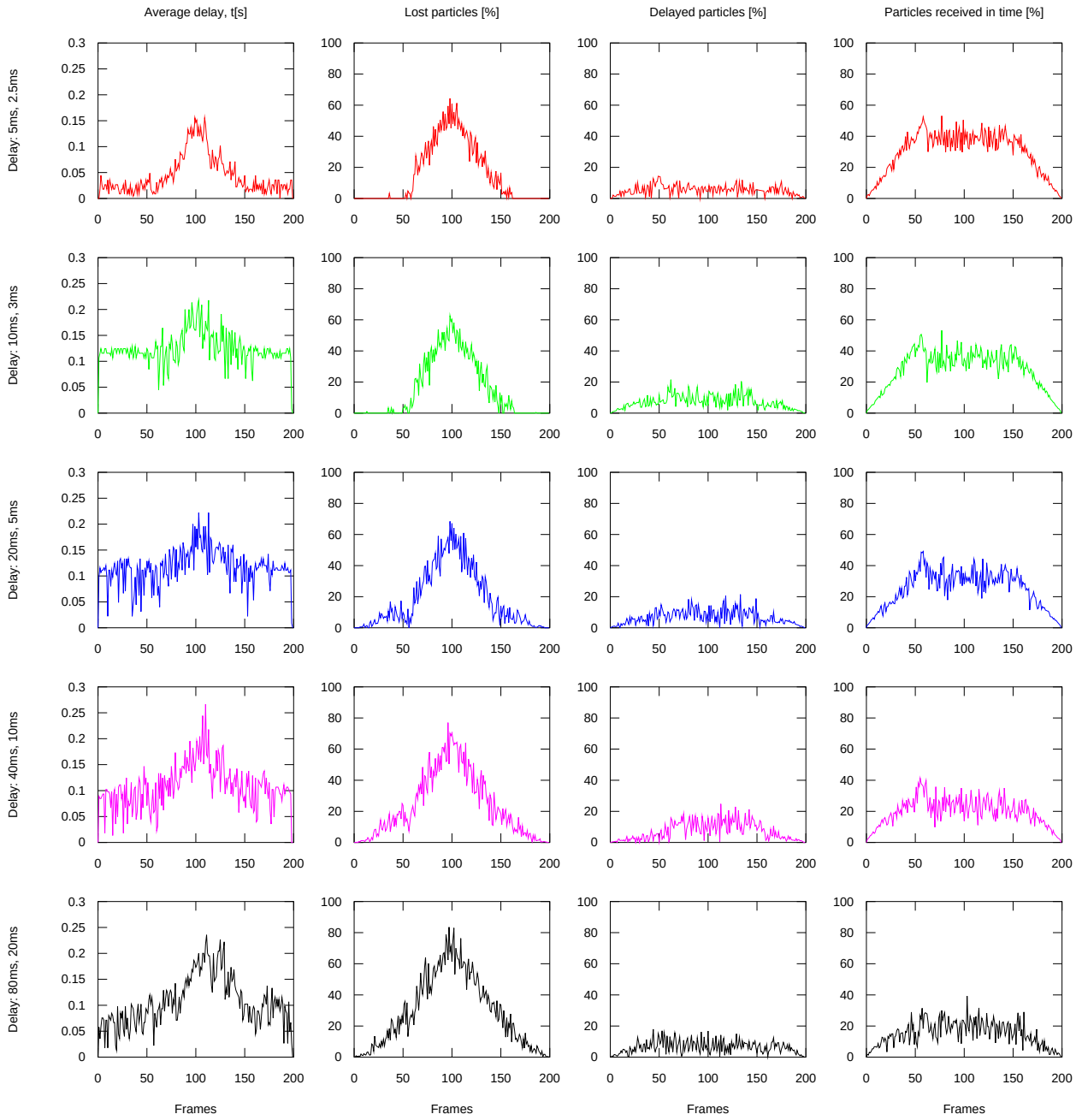
Kromě výše zmíněných transportních protokolů existuje celá řada dalších jako je například Reliable Datagram Protocol (RDP) [36] navržený pro distribuovaný operační systém Plan 9. Protokol RDP ani další transportní protokoly nebyly testovány, protože jejich testování by šlo nad rámec této disertační práce. Navíc je malá šance, že by se masivně rozšířili na hlavní operační systémy. Reálné nasazení jiného transportního protokolu než TCP a UDP navíc komplikuje špatná nebo nulová podpora u firewallů.

7.4 Aplikační protokoly

Z aplikačních protokolů byl otestován pouze původní a nový protokol Verse. V obou případech bylo potřeba upravit testovací aplikaci. Na virtualizovaném operačním systému běžel vždy Verse server a speciální Verse klient, který posílal serveru částicový systém. Server tento částicový systém následně přeposílal po upravené lince. Kombinace Verse serveru a tohoto klienta zastávala dohromady stejnou funkci jako měl server při testování transportních protokolů. Verse klient běžící na hostujícím operačním systému opět přijímal pakety poslané přes modifikovanou linku, porovnával je s vygenerovaným částicovým systémem a zobrazoval rozdíly.

7.4.1 Původní protokol Verse

Z výsledků měření původního protokolu Verse je patrné, že původní protokol má zpoždění doručení částic vyšší než bylo dosaženo při použití prostého UDP protokolu a spojitost pohybu částic byla taktéž nižší. Na druhou stranu protokol Verse je schopen přeposílat polohu ztracených částic, takže na konci simulace je částicový systém na serveru i klientovi ve stejné podobě. Stejnou funkcionalitu byl schopný zajistit i protokol TCP za cenu velkého zpoždění doručení částic. Výsledky měření původního protokolu jsou uvedeny na obrázku 7.9.



Obrázek 7.9: Výsledky měření za použití původního protokolu Verse

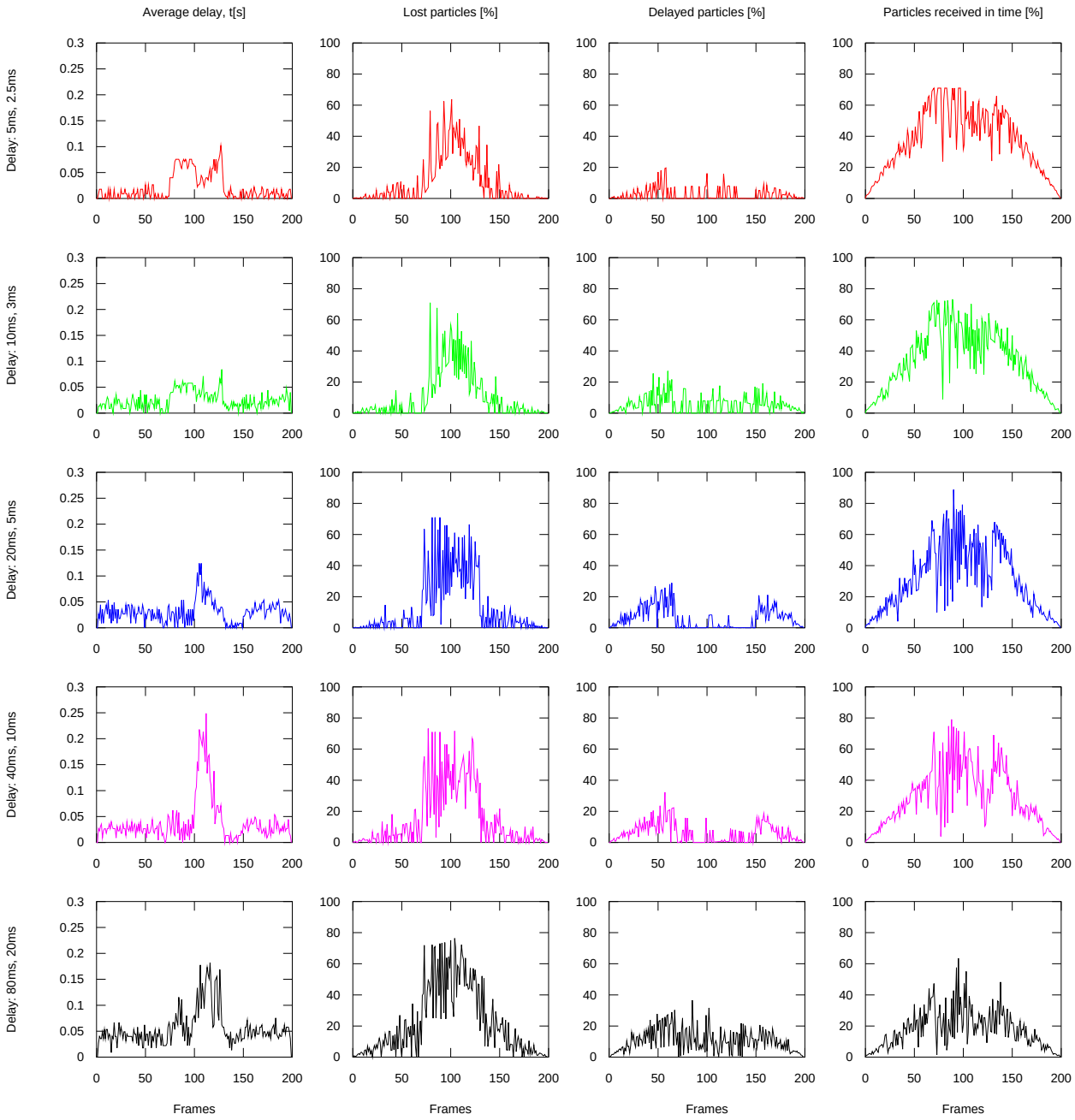
7.4.2 Nový protokol Verse

Při testování implementace nového protokolu Verse bylo ověřeno, že nový protokol je schopný efektivně přeposílat ztracené pakety, takže částicový systém je na konci přenosu na straně klienta také v konzistentním stavu. Měření zároveň ukázalo, že nový protokol Verse umožňuje přenášet polohu částic efektivněji než původní protokol. Docházelo v mnohem menší míře ke ztrátě paketů a částic, což lze přičíst efektivnějšímu využívání místa v paketu, jak bylo ukázáno na obrázku 5.21. Samotné výsledky měření nového protokolu jsou ukázány na obrázku 7.10, kde pro porovnání nebyla použita komprese příkazů.

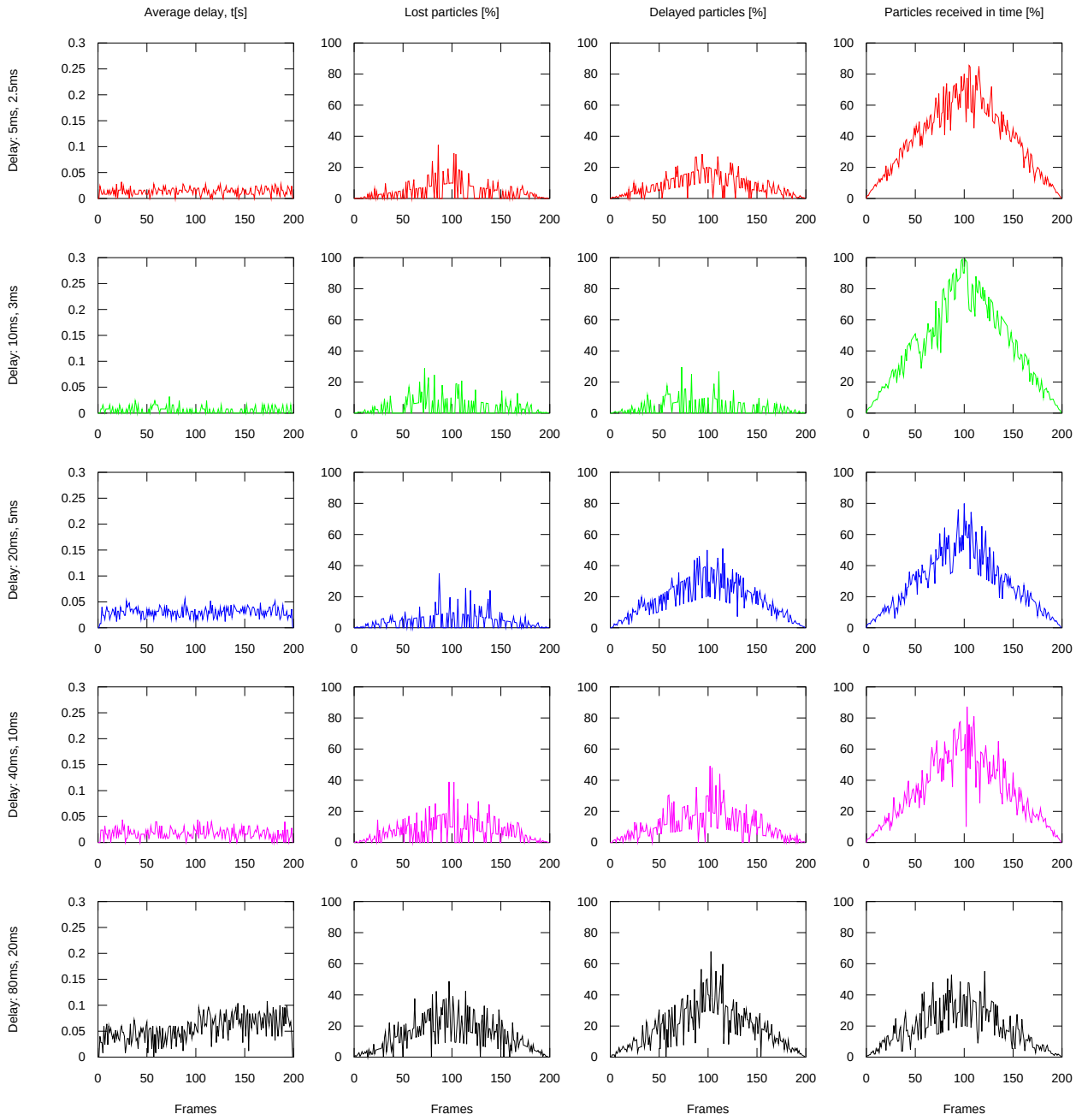
Když byla použita komprese příkazů, tak výsledný datový tok byl menší než nastavená šířka pásma. Díky tomu téměř nedocházelo ke ztrátám paketů a zpoždění částic bylo minimální. Z grafů na obrázku 7.11 je ovšem patrné, že se zvyšujícím se rozptylem zpoždění paketů docházelo k nárůstu ztráty částic, což bylo zapříčiněno změnou pořadí paketů. Při zvyšujícím se rozptylu zpoždění se zvyšovala i pravděpodobnost změny pořadí paketů. Jelikož se při implementaci protokolu použila jednodušší varianty, tak byly pakety se změněným pořadím klientem zahazovány.

Bohužel se nepodařilo provést měření přenosu za použití protokolu DTLS. Implementace protokolu Verse totiž využívá implementace protokolu DTLS z knihovny OpenSSL, která obsahuje chybu jež se projevuje náhodně při ztrátě paketu. Při výskytu této chyby jsou všechny následující pakety dekódovány špatně. Ve výsledku lze sice DTLS použít na lince, kde nedochází ke ztrátě paketu, ale už ho nebylo možné otestovat.

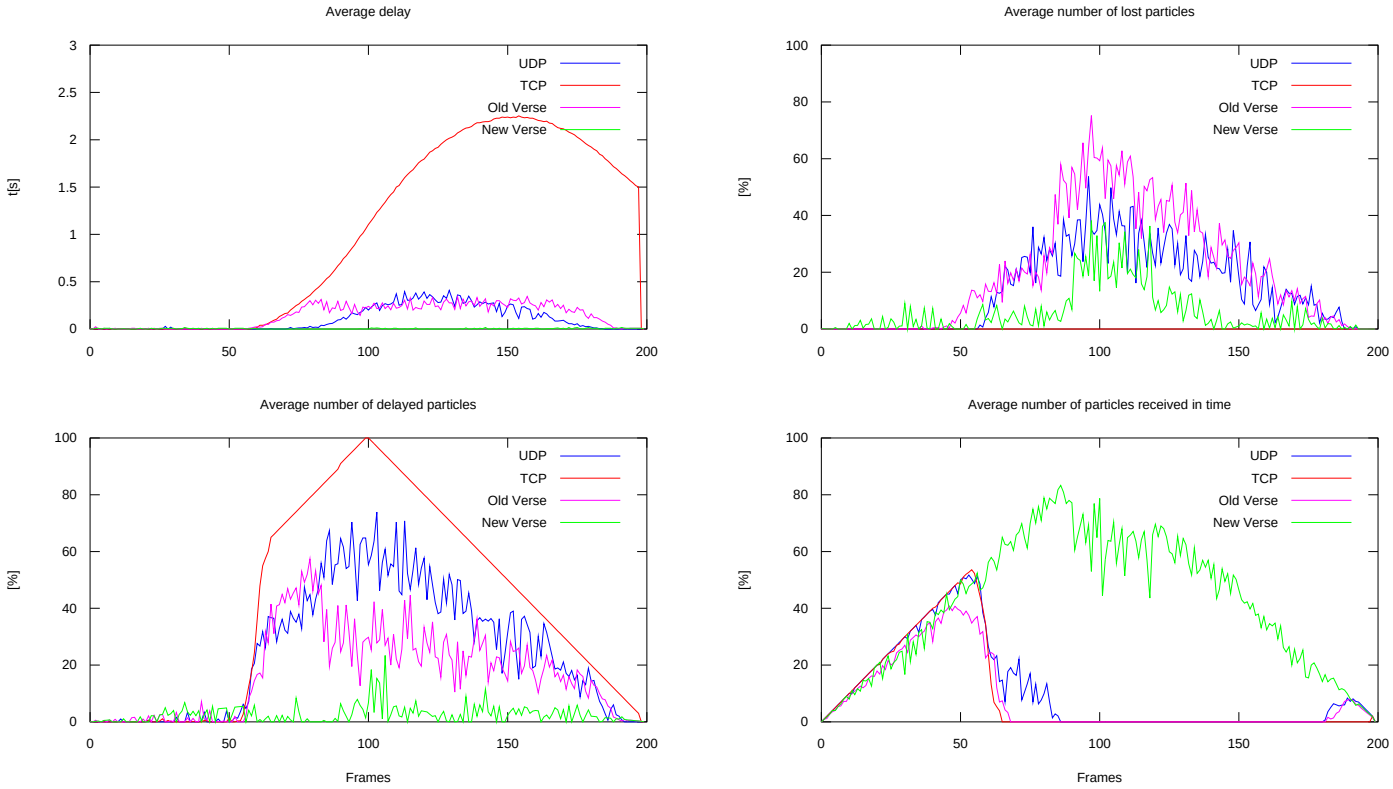
Výsledky experimentů v reálné síťovém prostředí jsou uvedeny na obrázku 7.12. Spojení, na kterém byly prováděny experimenty, mělo šířku pásma 1,9 Mb/s a průměrné zpoždění na lince bylo 5 ms. K experimentům byl použit částicový systém, který obsahoval 1000 částic. Z výsledků experimentů je patrné, že nový protokol Verse dává i v reálném síťovém provozu velmi dobré výsledky, které jsou způsobeny především efektivní kompresí přenášených dat.



Obrázek 7.10: Výsledky měření za použití nového protokolu Verse (bez komprese příkazů)



Obrázek 7.11: Výsledky měření za použití nového protokolu Verse (s kompresí příkazů)



Obrázek 7.12: Porovnání výsledků experimentálního měření v reálné síťovém provozu

Kapitola 8

Závěr

Tato disertační práce obsahuje popis specifikace nového protokolu Verse (kapitola 4) určeného pro real-timové sdílení dat v aplikacích sdílené virtuální reality. Specifikace částečně vychází z původního protokolu Verse (kapitola 3), ale celý protokol byl od základu přepracován s ohledem na větší bezpečnost, spohlivost a efektivitu přenosu dat. Specifikace obsahuje popis zahájení spojení, které kromě autentizace uživatele umožňuje provést dohadování o vlastnostech datové komunikace mezi klientem a serverem. Pro dosažení částečné spolehlivosti byl navržen nový robustnější a efektivnější resend mechanismus, který přeposílá pouze aktuální data a zaručuje efektivní využívání přenosových linek. Nové vlastnosti protokolu Verse byly prezentovány na konferenci BCONF 2010 [19].

Součástí specifikace je i popis nového datového modelu (kapitola 5), který sice klade na Verse klienty některé nové požadavky, ale zbytečné požadavky původního protokolu ruší. V konečném důsledku je možné provádět sdílení dat, které se starým protokolem nebylo možné. Navíc je možné nastavovat u sdílených dat přístupová práva, což původní protokol také neumožňoval.

Pro všechny důležité části nové specifikace byly vytvořeny verifikační modely v programovacím jazyku PROMELA a provedena jejich verifikovace pomocí nástroje Spin (kapitoly 4.5.3, 4.5.2 a 4.10). Výsledky verifikace resend mechanismu byly publikovány ve sborníku konference INTED 2009 [18].

Podstatná část specifikace byla implementována v programovacím jazyku C. Popis implementace (kapitola 6) obsahuje především obecné struktury a postupy, které umožňují efektivně implementovat nový protokol i v dalších programovacích jazycích.

Tato implementace byla použita pro měření v experimentálním prostředí, kdy byla otestovaná vhodnost jednotlivých transportních protokolů pro nový protokol Verse. Zároveň byly v tomto experimentálním prostředí provedeny testy původního a nového protokolu Verse, které ukázaly, že nový protokol

dává lepší výsledky. Výsledky těchto experimentů byly přijaty na konferenci WSCG 2011 [20].

Shrnutí přínosů k rozvoji vědního oboru

V práci je navržený resend mechanismus pro efektivní sdílení dat v aplikacích sdílené virtuální reality, kdy není vyžadován přenos dat s úplnou spolehlivostí, ale jsou kladeny požadavky na nízké latence doručení aktuálních dat. Navržené algoritmy navíc umožňují stanovit priority sdíleným datům a tak efektivně zvýšit šanci jejich včasného doručení.

Shrnutí přínosů pro praxi

Navržený protokol umožňuje sdílet data mezi grafickými aplikacemi bezpečněji, spolehlivěji a hlavně efektivněji než původní protokol Verse. Při programování nových Verse klientů je možné efektivně využít vícevláknové charakteru knihovny a vícevláknová implementace Verse serveru umožňuje lepší škálování.

Další práce a experimenty

Další práce by měla spočívat především v dokončení implementace celé specifikace a opravení chybné implementace DTLS v knihovně OpenSSL, která znemožňuje praktické nasazení zabezpečeného přenosu v praxi. Plnohodnotná implementace DTLS protokolu by měla být následně otestována především s ohledem na velké datové toky a zatížení Verse serveru. Další rozšíření specifikace by se mělo týkat především Congestion Control (kapitola 4.12) v kombinaci s dohadováním o FPS (kapitola 4.5.3), protože posílání paketů v pravidelných intervalech odpovídajících FPS Verse klienta se jeví jako další možný způsob jak zefektivnit přenos dat.

Další rozšíření specifikace by se mělo týkat pravidel sdílení dat na serveru pomocí DED (kapitola 4.5.3). Grafické aplikace by měly mít jednak možnost definovat si vlastní pravidla, která jsou jim ušita na míru a zároveň by mělo být vytvořena sada pravidel, která by umožňovala sdílet data mezi různými grafickými aplikacemi. V neposlední řadě by měla být provedena opětovná implementace nového protokolu Verse do programu Blender.

Literatura

[1] Al-Regib, G., and Altunbasak, Y. 3TP: 3-D models transport protocol. In *Web3D '04: Proceedings of the ninth international conference on 3D Web technology* (New York, NY, USA, 2004), ACM, pp. 155–162.

[2] Allman, M., Paxson, V., and Stevens, W. TCP Congestion Control. RFC 2581, IETF, apr 1999. `http://www.ietf.org/rfc/rfc2581.txt`, Obsoleted by RFC 5681, updated by RFC 3390.

[3] Arnaud, R., and Barnes, M. C. *Collada: Sailing the Gulf of 3d Digital Content Creation*. AK Peters Ltd, 2006.

[4] Ayuso, P. N. Netfilter/iptables. `http://www.netfilter.org/`, 2010.

[5] Bennett, J. C. R., Partridge, C., and Shectman, N. Packet reordering is not pathological network behavior. *IEEE/ACM Trans. Netw. 7*, 6 (December 1999), 789–798.

[6] Berners-Lee, T., Masinter, L., and McCahill, M. Uniform Resource Locators (URL). RFC 1738, IETF, dec 1994. `http://www.ietf.org/rfc/rfc1738.txt`, Obsoleted by RFCs 4248, 4266, updated by RFCs 1808, 2368, 2396, 3986.

[7] Boulanger, J.-S., Kienzle, J., and Verbrugge, C. Comparing interest management algorithms for massively multiplayer games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games* (New York, NY, USA, 2006), NetGames '06, ACM.

[8] Bouras, C., Giannaka, E., and Tsiatsos, T. Partitioning of Distributed Virtual Environments Based on Objects' Attributes. In *Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications* (Washington, DC, USA, 2007), DS-RT '07, IEEE Computer Society, pp. 72–75.

[9] Brink, E., Steenberg, E., and Svensson, G. The Verse Networked 3D Graphics Platform. In *SIGRAD 2006, The Annual SIGRAD Conference* (Linköping, Sweden, 2006), H. Gustavsson, Ed., SIGRAD, pp. 44–48.

[10] Cruz-Neira, C., Sandin, D. J., DeFanti, T. A., Kenyon, R. V., and Hart, J. C. The CAVE: audio visual experience automatic virtual environment. *Commun. ACM 35*, 6 (June 1992), 64–72.

[11] Dierks, T., and Allen, C. The TLS Protocol Version 1.0. RFC 2246, IETF, jan 1999. `http://www.ietf.org/rfc/rfc2246.txt`, Obsoleted by RFC 4346, updated by RFCs 3546, 5746.

[12] Floyd, S., and Kohler, E. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control. RFC 4341, IETF, mar 2006. `http://www.ietf.org/rfc/rfc4341.txt`.

[13] Floyd, S., Kohler, E., and Padhye, J. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC). RFC 4342, IETF, mar 2006. `http://www.ietf.org/rfc/rfc4342.txt`, Updated by RFC 5348.

[14] Group, A.-V. T. W., Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. RTP: A Transport Protocol for Real-Time Applications. RFC 1889, IETF, jan 1996. `http://www.ietf.org/rfc/rfc1889.txt`, Obsoleted by RFC 3550.

[15] Harcsik, S., Petlund, A., Griwodz, C., and Halvorsen, P. Latency evaluation of networking mechanisms for game traffic. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games* (New York, NY, USA, 2007), NetGames '07, ACM, pp. 129–134.

[16] Hemminger, S. Network Emulation with NetEm. In *Linux Conf Au* (April 2005).

[17] Hnidek, J. Integration of Verse protocol to Blender. Interantional Blender Conference, Blender Foundation, Amsterdam, the Netherlands, October 2005.

[18] Hnidek, J. Resend Mechanism for Reliable Datagram Protocol. *Annual Edition of the International Technology. Education and Development Conference (INTED)* (2009).

[19] Hnidek, J. Introduction of New Verse Protocol. Interantional Blender Conference, Stichting Blender Foundation, Amsterdam, the Netherlands, October 2010.

[20] Hnidek, J. Network Protocols for Applications of Shared Virtual Reality. *19th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision Communication Papers Proceedings* (2011).

[21] Holzmann, G. J. *Design and validation of computer protocols.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[22] Jacobson, V. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev. 18*, 4 (August 1988), 314–329.

[23] Kempf, J., Chander, A., and Jo, M. Optimizing avatar environmental update in shared virtual reality environments. In *Proceedings of the First International Conference on Immersive Telecommunications* (ICST, Brussels, Belgium, Belgium, 2007), ImmersCom '07, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 1–1.

[24] Keshav, S. *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[25] Kohler, E., Handley, M., and Floyd, S. RFC 4340: Datagram Congestion Control Protocol (DCCP). RFC 4340, IETF, mar 2006. `http://www.ietf.org/rfc/rfc4340.txt`, Updated by RFCs 5595, 5596.

[26] Lui, J. C. S., and Chan, M. F. An Efficient Partitioning Algorithm for Distributed Virtual Environment Systems. *IEEE Trans. Parallel Distrib. Syst. 13*, 3 (March 2002), 193–211.

[27] Mills, D., Martin, J., Burbank, J., and Kasch, W. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905, IETF, jun 2010. `http://www.ietf.org/rfc/rfc4341.txt`.

[28] Modadugu, N., and Rescorla, E. The Design and Implementation of Datagram TLS. In *In Proc. NDSS* (2004).

[29] Postel, J. RFC 768: User Datagram Protocol. RFC 768, IETF, aug 1980. http://www.ietf.org/rfc/rfc768.txt.

[30] Postel, J. RFC 793: Transmission Control Protocol. RFC 793, IETF, sep 1981. http://www.ietf.org/rfc/rfc793.txt, Updated by RFCs 1122, 3168.

[31] Rescorla, E., and Modadugu, N. Datagram Transport Layer Security. RFC 4347, IETF, apr 2006. http://www.ietf.org/rfc/rfc4347.txt, Updated by RFC 5746.

[32] Steenberg, E. Brink, E. The Verse Specification. http://verse.blender.org, 2007.

[33] Stevens, W. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001, IETF, jan 1997. http://www.ietf.org/rfc/rfc2001.txt, Obsoleted by RFC 2581.

[34] Stewart, R. RFC 4960: Stream Control Transmission Protocol. RFC 4960, IETF, sep 2007. http://www.ietf.org/rfc/rfc4960.txt.

[35] The Linux Foundation. Networking ToDo List. Tech. rep., The Linux Foundation, 1796 18th Street, Suite C, San Francisco, CA 94107, 2010. http://www.linuxfoundation.org/collaborate/workgroups/networking/todo.

[36] Velten, D., Hinden, R. M., and Sax, J. Reliable Data Protocol. RFC 908, IETF, jul 1984. http://www.ietf.org/rfc/rfc908.txt, Updated by RFC 1151.

[37] Yergeau, F. UTF-8, a transformation format of ISO 10646. RFC 3629, IETF, nov 2003. http://www.ietf.org/rfc/rfc3629.txt.

Kapitola 9

Přílohy verifikačních modelů

Verifikace autentifikace uživatele

```
1  /**
2   *
3   * This PROMELA program simulates and helps to verify the user
4   * authentication used in Verse protocol over TCP/TLS connection.
5   *
6   * author: Jiri Hnidek <jiri.hnidek@tul.cz>
7   */
8
9  #define UNVALID_USER          0
10 #define VALID_USER            1
11
12 #define UNVALID_DATA          0
13 #define VALID_DATA            1
14
15 #define AUTH_FAIL             0
16 #define AUTH_SUCC             1
17
18 #define AUTH_METH_RESV        0
19 #define AUTH_METH_NONE        1
20 #define AUTH_METH_DATA        2
21
22 #define MAX_AUTH_ATTEMPTS     3
23
24 #define NOT_EVIL              0
25 #define EVIL                  1
26
27 #define UNSTABLE              0
28 #define STABLE                1
29
30 byte user, data;
31
32 /**
33  * This process simulates the Verse client during user authentication. Every
34  * process should be in the state named 'end*' at the end of the run.
35  */
36 proctype Client(chan in; chan out)
37 {
38     byte cmd_type;
39     byte auth_meth;
40     bit stable;
```

```
41
42      /* This client could be stable or potentionaly unstable */
43      do
44      :: stable = STABLE -> { printf("Client: stable\n"); break; };
45      :: stable = UNSTABLE -> { printf("Client: unstable\n"); break; };
46      od;
47
48      /* Random pick of valid and unvalid username and authentication data */
49      do
50          :: user = VALID_USER; data = VALID_DATA; break;
51          :: user = VALID_USER; data = UNVALID_DATA; break;
52          :: user = UNVALID_USER; data = UNVALID_DATA; break;
53          :: user = UNVALID_USER; data = VALID_DATA; break;
54      od;
55
56  /* First initial state */
57  closed:
58      printf("Client: CLOSED\n");
59      skip;
60
61  /* Second state; client sends username with authenticate method type NONE */
62  user_auth_none:
63      printf("Client: USER_AUTH none\n");
64      do
65      :: out!user,AUTH_METH_NONE,0;
66      :: in?cmd_type,auth_meth;
67          if
68          /* Server sent list of supported authenticated methods */
69          :: (cmd_type==AUTH_FAIL &&
70              auth_meth==AUTH_METH_DATA) -> { goto user_auth_data; };
71          /* Server should not close connection, when username is not valid */
72          :: (cmd_type==AUTH_FAIL &&
73              auth_meth==AUTH_METH_RESV) -> { goto end_auth_fail; };
74          /* Server should not authenticate user only due to username */
75          :: (cmd_type==AUTH_SUCC) -> { goto end_auth_succ; };
76          :: else -> { goto end_auth_fail; };
77          fi;
78      :: (stable==UNSTABLE) -> { goto end_auth_fail; };
79      :: timeout -> goto end_auth_fail;
80      od;
81
82  /* Third state; client sends username with authentication data */
83  user_auth_data:
84      printf("Client: USER_AUTH data\n");
85      do
86      :: out!user,AUTH_METH_DATA,data;
87      /* Try to receive some response from server */
88      :: in?cmd_type,auth_meth;
89          if
90          /* Client was authenticated. */
91          :: (cmd_type == AUTH_SUCC) -> { goto end_auth_succ; };
92          /* Authentication of client failed. */
93          :: (cmd_type == AUTH_FAIL &&
94              auth_meth == AUTH_METH_RESV) -> { goto end_auth_fail; };
95          /* Authentication of client failed, but server gave the client one
96             more chance. */
97          :: (cmd_type == AUTH_FAIL &&
98              auth_meth == AUTH_METH_DATA) -> { goto user_auth_data; };
99          /* Other combinations are not allowed too. */
100         :: else -> { goto end_auth_fail; };
101         fi;
102     :: (stable==UNSTABLE) -> { goto end_auth_fail; };
```

```
103         :: timeout -> { goto end_auth_fail; };
104         od;
105
106     /* Valid and unsuccessful end state */
107     end_auth_fail:
108         printf("Client: FAILURE\n");
109
110         /* Wait for server process */
111         do
112         :: in?cmd_type,auth_meth
113         :: timeout -> { goto end; };
114         od;
115
116     /* Valid and successful end state */
117     end_auth_succ:
118         printf("Client: SUCCESS\n");
119
120     /* Valid end state */
121     end:
122         skip;
123     }
124
125     /**
126      * This process simulates the Verse server during user authentication.
127      */
128     proctype Server(chan in; chan out)
129     {
130         byte user_name, init_user_name;
131         byte auth_meth;
132         byte auth_data;
133         byte attempts = 0;
134         bit stable;
135
136         /* The server could be stable or potentionaly unstable */
137         do
138         :: stable = STABLE -> { printf("Server: stable\n"); break; };
139         :: stable = UNSTABLE -> { printf("Server: unstable\n"); break; };
140         od;
141
142     /* First server state; server listen for client requests. */
143     listen:
144         printf("Server: LISTEN\n");
145         do
146         :: in?user_name,auth_meth,auth_data;
147             if
148             /* First authentication attempt with method NONE -> send list of
149                allowed methods. */
150             :: (auth_meth == AUTH_METH_NONE) -> atomic {
151                     init_user_name = user_name;
152                     goto respond_user_auth;
153                 };
154             /* Note: Real server should not authenticate user only
155                due to username. */
156             :: (user_name == VALID_USER) -> { goto end_auth_succ; };
157             /* Other combinations are not allowed. */
158             :: else -> { goto end_auth_fail; }
159             fi;
160         :: (stable==UNSTABLE) -> goto end_auth_fail;
161         :: timeout -> goto end_auth_fail;
162         od;
163
164     /* Second server state; server sends list of supported methods and then listen
```

```
165     for combination of username and authentication data. */
166 respond_user_auth:
167     printf("Server: RESPOND user_auth\n");
168     out!AUTH_FAIL,AUTH_METH_DATA;
169     do
170     :: in?user_name,auth_meth,auth_data;
171         if
172         :: (attempts < MAX_AUTH_ATTEMPTS &&
173             user_name == init_user_name &&
174             user_name == VALID_USER &&
175             auth_meth == AUTH_METH_DATA &&
176             auth_data == VALID_DATA) -> { goto end_auth_succ; };
177         :: (attempts < MAX_AUTH_ATTEMPTS &&
178             user_name == init_user_name &&
179             user_name == VALID_USER &&
180             auth_meth == AUTH_METH_DATA &&
181             auth_data != VALID_DATA) -> atomic {
182                 attempts++;
183                 goto respond_user_auth;
184         };
185         :: (attempts < MAX_AUTH_ATTEMPTS &&
186             user_name == init_user_name &&
187             user_name != VALID_USER &&
188             auth_meth == AUTH_METH_DATA) -> atomic {
189                 attempts++;
190                 goto respond_user_auth;
191         };
192         :: (attempts >= MAX_AUTH_ATTEMPTS) -> { goto end_auth_fail; };
193         :: (user_name != init_user_name) -> { goto end_auth_fail; };
194         :: else -> { goto end_auth_fail; };
195         fi;
196     :: (stable==UNSTABLE) -> goto end_auth_fail;
197     :: timeout -> goto end_auth_fail;
198     od;
199
200 /* Valid and successful end state */
201 end_auth_succ:
202     printf("Server: SUCCESS\n");
203     out!AUTH_SUCC,0;
204     goto end;
205
206 /* Valid and unsuccessful end state */
207 end_auth_fail:
208     printf("Server: FAILED\n");
209     do
210     :: in?user_name,auth_meth,auth_data;
211     :: timeout -> goto end;
212     od;
213
214 /* Valid end state */
215 end:
216     skip;
217 }
218
219 /**
220  * Initial process, where communication channels are created and client and
221  * server processes are started. Channels have zero length. It means that
222  * channels are synchronous.
223  */
224 init
225 {
226     /* user_name, auth_type, auth_data */
```

```
227       chan q1 = [0] of {bit, byte, bit};
228       /* cmd_type, auth_meth */
229       chan q2 = [0] of {bit, byte};
230
231       atomic{run Server(q1, q2); run Client(q2, q1);};
232   }
```

Výsledky verifikace autentifikace uživatele Verifikace byla vždy provedena nejprve pro vyhledání nekorektních koncových stavů a následně byla verifikace provedena pro nalezení případných nevyvíjejících se cyklů.

```
$ spin -a user_auth.pml
$ gcc -o user_auth -DSAFETY pan.c
$ ./user_auth

(Spin Version 5.2.5 -- 17 April 2010)
+ Partial Order Reduction

Full statespace search for:
never claim          - (none specified)
assertion violations +
cycle checks        - (disabled by -DSAFETY)
invalid end states +

State-vector 52 byte, depth reached 68, errors: 0
     1773 states, stored
      244 states, matched
     2017 transitions (= stored+matched)
        1 atomic steps
hash conflicts:         1 (resolved)

    2.598 memory usage (Mbyte)

unreached in proctype Client
(0 of 90 states)
unreached in proctype Server
(0 of 76 states)
unreached in proctype :init:
(0 of 4 states)

pan: elapsed time 0 seconds
```

```
$ spin -a user_auth.pml
$ gcc -o user_auth -DNP pan.c
$ ./user_auth -l

(Spin Version 5.2.5 -- 17 April 2010)
+ Partial Order Reduction

Full statespace search for:
never claim          +
assertion violations + (if within scope of claim)
non-progress cycles  + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 60 byte, depth reached 113, errors: 0
     4541 states, stored (6623 visited)
     5218 states, matched
    11841 transitions (= visited+matched)
      602 atomic steps
hash conflicts:         8 (resolved)

Stats on memory usage (in Megabytes):
    0.329 equivalent memory usage for states (stored*(State-vector + overhead))
    0.372 actual memory usage for states (unsuccessful compression: 113.09%)
          state-vector as stored = 70 byte + 16 byte overhead
    2.000 memory used for hash table (-w19)
    0.305 memory used for DFS stack (-m10000)
    2.598 total actual memory usage

unreached in proctype Client
(0 of 96 states)
unreached in proctype Server
(0 of 76 states)
unreached in proctype :init:
(0 of 4 states)

pan: elapsed time 0.01 seconds
```

```
1  /**
2   *
3   * This PROMELA program simulates and helps to verify the negotiation of new
4   * datagram connection between Verse client and Verse server.
5   *
6   * author: Jiri Hnidek <jiri.hnidek@tul.cz>
7   */
8
9  mtype = {none, change_l, change_r, confirm_l, confirm_r}
10
11 #define NONE_URL     0
12 #define VALID_URL    1
13 #define UNVALID_URL  2
14
15 #define UNSTABLE     0
16 #define STABLE       1
17
18 /**
19  * This process simulates the Verse client during negotiation of new datagram
20  * connection.
21  */
22 proctype Client(chan in; chan out)
23 {
24     mtype cmd1, cmd2;
25     byte url1, url2;
26     bit stable;
27
28     /* This client could be stable or potentionaly unstable */
29     do
30     :: stable = STABLE -> { printf("Client: stable\n"); break; };
31     :: stable = UNSTABLE -> { printf("Client: unstable\n"); break; };
32     od;
33
34 authenticated:
35     printf("Client: AUTHENTICATED\n");
36
37     do
38     :: (1) -> break;
39     :: (stable==UNSTABLE) -> { goto end_failure; };
40     od;
41
42 propose_url:
43     printf("Client: PROPOSE_URL\n");
44     out!change_r, UNVALID_URL, none, 0;
45
46     /* Wait for servere response */
47     do
48     :: in?cmd1, url1, cmd2, url2;
49         if
50         :: (cmd1==confirm_r &&
51             url1==NONE_URL &&
52             cmd2==change_l &&
53             url2==VALID_URL) -> { goto try_url; };
54         :: timeout -> { goto end_failure; };
55         fi;
56     :: (stable==UNSTABLE) -> { goto end_failure; };
57     :: timeout -> { goto end_failure; };
58     od;
59
60 try_url:
```

```
61      printf("Client: TRY_URL\n");
62      /* Random pick result of connection */
63      do
64      :: url1=VALID_URL ->
65          { printf("Client: Dgram handshake successful\n"); break; };
66      :: url1=NONE_URL ->
67          { printf("Client: Dgram handshake failed\n"); break; };
68      od;
69
70      do
71      :: (1) -> break;
72      :: (stable==UNSTABLE) -> { goto end_failure; };
73      od;
74
75  negotiate_url:
76      printf("Client: NEGOTIATE_URL\n");
77      out!confirm_l, url1, none, 0;
78      if
79      :: (url1==VALID_URL) -> { goto end_success; };
80      :: (url1==NONE_URL) -> { goto end_failure; };
81      fi;
82
83  end_failure:
84      printf("Client: FAILURE\n");
85
86      /* Wait for other proccess */
87      do
88      :: in?cmd1, url1, cmd2, url2;
89      :: timeout -> { goto end; };
90      od;
91
92  end_success:
93      printf("Client: SUCCESS\n");
94
95  end:
96      skip;
97  }
98
99
100
101 /**
102  * This process simulates the Verse server during negotiation of new
103  * datagram connection.
104  */
105 proctype Server(chan in; chan out)
106 {
107     mtype cmd1, cmd2;
108     byte url1, url2;
109     bit stable;
110
111     /* This server could be stable or potentially unstable */
112     do
113     :: stable = STABLE -> { printf("Server: stable\n"); break; };
114     :: stable = UNSTABLE -> { printf("Server: unstable\n"); break; };
115     od;
116
117 authenticated:
118     printf("Server: AUTHENTICATED\n");
119
120     /* Wait for URL proposal */
121     do
122     :: in?cmd1, url1, cmd2, url2;
```

```
123             if
124             :: (cmd1==change_r &&
125                 url1==UNVALID_URL) -> { goto create_url; };
126             :: else  -> { goto end_failure; };
127             fi;
128         :: (stable==UNSTABLE) -> { goto end_failure; };
129         :: timeout -> { goto end_failure; };
130         od;
131
132     create_url:
133         printf("Server: CREATE_URL\n");
134
135         /* Random pick result of creating new dgram socket */
136         do
137         :: url2=VALID_URL ->
138             { printf("Server: Open dgram port successful\n"); break; };
139         :: url2=NONE_URL ->
140             { printf("Server: Open dgram port failed\n"); break; };
141         od;
142
143         out!confirm_r, NONE_URL, change_l, url2;
144         if
145         :: (url2==NONE_URL) -> { goto end_failure; };
146         :: else -> skip;
147         fi;
148
149     negotiate_url:
150         /* Wait for URL confirmation */
151         do
152         :: in?cmd1, url1, cmd2, url2;
153             if
154             :: (cmd1==confirm_l &&
155                 url1==VALID_URL) -> { goto end_success; };
156             :: else -> { goto end_failure; };
157             fi;
158         :: (stable==UNSTABLE) -> { goto end_failure; };
159         :: timeout -> { goto end_failure; };
160         od;
161
162     end_failure:
163         printf("Server: FAILURE\n");
164
165         /* Wait for other proccess */
166         do
167         :: in?cmd1, url1, cmd2, url2;
168         :: timeout -> { goto end; };
169         od;
170
171     end_success:
172         printf("Server: SUCCESS\n");
173
174     end:
175         skip;
176     }
177
178     /**
179      * Initial process, where communication channels are created and client and
180      * server processes are started. Channels have zero length. It means that
181      * channels are synchronous.
182      */
183     init
184     {
```

```
185       /* cmd_type, URL, cmd_type, URL */
186       chan q1 = [0] of {mtype, byte, mtype, byte};
187       /* cmd_type, URL, cmd_type, URL */
188       chan q2 = [0] of {mtype, byte, mtype, byte};
189
190       atomic{run Server(q1, q2); run Client(q2, q1);};
191   }
```

Výsledky verifikace dohadování o novém datagramovém spojení

```
$ spin -a host_neg.pml
$ gcc -o host_neg -DSAFETY pan.c
$ ./host_neg

(Spin Version 5.2.5 -- 17 April 2010)
+ Partial Order Reduction

Full statespace search for:
never claim          - (none specified)
assertion violations +
cycle checks        - (disabled by -DSAFETY)
invalid end states +

State-vector 56 byte, depth reached 43, errors: 0
      517 states, stored
       56 states, matched
      573 transitions (= stored+matched)
        1 atomic steps
hash conflicts:         0 (resolved)

    2.501 memory usage (Mbyte)

unreached in proctype Client
(0 of 81 states)
unreached in proctype Server
(0 of 79 states)
unreached in proctype :init:
(0 of 4 states)

pan: elapsed time 0.01 seconds
```

```
$ spin -a host_neg.pml
$ gcc -o host_neg -DNP pan.c
$ ./host_neg -l

(Spin Version 5.2.5 -- 17 April 2010)
+ Partial Order Reduction

Full statespace search for:
never claim          +
assertion violations + (if within scope of claim)
non-progress cycles  + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 60 byte, depth reached 83, errors: 0
     1033 states, stored (1549 visited)
     1255 states, matched
     2804 transitions (= visited+matched)
        2 atomic steps
hash conflicts:         0 (resolved)

    2.501 memory usage (Mbyte)

unreached in proctype Client
(0 of 81 states)
unreached in proctype Server
(0 of 79 states)
unreached in proctype :init:
(0 of 4 states)

pan: elapsed time 0.01 seconds
```

```
1  /**
2   *
3   * This PROMELA program simulates and helps to verify the datagrame
4   * connection of Verse protocol.
5   *
6   * author: Jiri Hnidek <jiri.hnidek@tul.cz>
7   *
8   * This verification model works only with parameter -m, because spin does not
9   * block, when channel is full; instead, message is dropped. This parameter
10  * has to be used during simulation as well for generating verifier!
11  *
12  */
13
14 /* Queue length */
15 #define QLEN 3
16 /* Maximal number of connection/teardown attempts */
17 #define MAX_ATTEMPTS 3
18 /* Number of payload packets sent by client */
19 #define N 2
20 /* Consider peer as death after 10 sent payload packets and no response
21  * from peer */
22 #define TIMEOUT 10
23
24 proctype Client(chan in; chan out)
25 {
26     /* Receive bit flags */
27     bit rPAY=0, rACK=0, rSYN=0, rFIN=0;
28     /* Send bit flags */
29     bit sPAY=0, sACK=0, sSYN=0, sFIN=0;
30     /* Receive PayID, AckID and MSG */
31     short rPAY_ID=0, rACK_ID=0, rMSG=0;
32     /* Send PayID, AckID and MSG */
33     short sPAY_ID=0, sACK_ID=0, sMSG=0;
34
35     /* Temporary variables */
36     short conn=0;    /* Number of connection attempts */
37     short f_pay;     /* Number of first payload packet */
38     short timer=0;
39     bit want_close=0;
40
41     /* Handshake */
42 closed:
43     printf("Client: CLOSED\n");
44
45     /* First phase of handshake */
46 request:
47     printf("Client: REQUEST\n");
48     sPAY=1; sSYN=1;
49 request_again:
50     if
51     /* Max number of connection attemps reached. */
52     :: (conn >= MAX_ATTEMPTS) -> { goto end; };
53     :: else -> atomic {
54             out!sPAY,sACK,sSYN,sFIN,sPAY_ID,sACK_ID,sMSG;
55             conn++;
56         };
57     fi;
58
59     do
60     :: (nempty(in)) -> atomic {
```

```
61          in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG ->
62              if
63              :: (rPAY==1 && rACK==1 && rSYN==1 && rMSG==sPAY_ID) -> { goto partopen; };
64              /* Simulation of packet loss */
65              :: (1) -> printf("Packet loss\n");
66              fi;
67          };
68      :: timeout -> atomic { goto request_again; };
69      od;
70
71      /* Second phase of handshake */
72  partopen:
73      printf("Client: PARTOPEN\n");
74      sSYN=0; sACK=1;
75      sPAY_ID=1;
76      sMSG=rPAY_ID;
77      conn=0;
78  partopen_again:
79      if
80      :: (conn >= MAX_ATTEMPTS) -> atomic {
81              printf("Client: max attempts reached\n");
82              goto end_failure;
83          };
84      :: else -> atomic {
85              out!sPAY,sACK,sSYN,sFIN,sPAY_ID,sACK_ID,sMSG;
86              conn++;
87          };
88      fi;
89
90      do
91      :: (nempty(in)) -> atomic {
92              in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG;
93              if
94              :: (rPAY==1 && rACK==1 && rMSG==sPAY_ID) -> { goto open; };
95              /* Simulation of packet loss */
96              :: (1) -> printf("Packet loss/drop\n");
97              fi;
98          };
99      :: timeout -> goto partopen_again;
100     od;
101
102     /* Handshake finished, normal communication */
103 open:
104     printf("Client: OPEN\n");
105     sMSG=rPAY_ID;
106
107     do
108     :: timer++;
109         if
110         /* Send payload packet */
111         :: (nfull(out) && timer<TIMEOUT) -> atomic {
112             out!sPAY,sACK,sSYN,sFIN,sPAY_ID,sACK_ID,sMSG; sPAY_ID++;
113             };
114         /* Receive packet */
115         :: (nempty(in) && timer<TIMEOUT) -> atomic {
116             in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG ->
117                 timer=0;
118                 do
119                 :: (rACK==1) -> {
120                         rACK=0;
121                         if
122                         :: (rMSG>=N) -> { goto closing; };
```

```
123                         :: else -> skip;
124                         fi;
125                     };
126                 :: (rFIN==1) -> { rFIN=0; sFIN=1; want_close=1; };
127                 :: else -> {
128                     if
129                     :: (want_close==1) -> { sMSG=rPAY_ID; goto closing; };
130                     :: else -> skip;
131                     fi;
132                     if
133                     :: (rPAY==1) -> {
134                             sMSG=rPAY_ID;
135                             sACK=1;
136                             /* Send acknowledgment packet */
137                             out!0,sACK,sSYN,sFIN,0,sACK_ID,sMSG;
138                             sACK_ID++;
139                         };
140                     :: else -> skip;
141                     fi;
142                     break;
143                     };
144                 od;
145             };
146         /* Simulation of packet loss */
147         :: (nempty(in) && timer<TIMEOUT) -> atomic {
148                 in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG;
149                 printf("Packet loss\n");
150             };
151         /* Other cases */
152         :: (empty(in) && full(out) && timer<TIMEOUT) -> skip;
153         /* No response from server */
154         :: (timer>=TIMEOUT) -> goto end_failure;
155         fi;
156     od;
157
158     /* Teardown */
159 closing:
160     printf("Client: CLOSING\n");
161     sPAY=1; sACK=1; sSYN=0; sFIN=1;
162     conn=0;
163 closing_again:
164     if
165     /* Server died during teardown, end. */
166     :: (conn >= MAX_ATTEMPTS) -> atomic {
167             printf("Client: max attempts reached\n");
168             goto end_failure;
169         };
170     :: else -> atomic {
171             conn++;
172             out!sPAY,sACK,sSYN,sFIN,sPAY_ID,sACK_ID,sMSG;
173         };
174     fi;
175
176     do
177     /* Receive packet */
178     :: (nempty(in)) -> atomic {
179         in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG;
180         if
181         :: (rFIN==1 && rACK==1 && rMSG==sPAY_ID) -> { goto end_closed; };
182         /* Simulation of packet loss */
183         :: (1) -> printf("Packet loss/drop\n");
184         fi;
```

```
185             };
186         :: timeout -> atomic { goto closing_again; };
187         od;
188
189     end_closed:
190         printf("Client: CLOSED\n");
191         goto end_success;
192
193     end_failure:
194         printf("Client: failure\n");
195         do
196         :: in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG;
197         :: timeout -> goto end;
198         od;
199
200     end_success:
201         printf("Client: success\n");
202         do
203         :: in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG;
204         :: timeout -> goto end;
205         od;
206
207     end:
208         skip;
209     }
210
211     proctype Server(chan in; chan out)
212     {
213         /* Receive bit flags */
214         bit rPAY=0, rACK=0, rSYN=0, rFIN=0;
215         /* Send bit flags */
216         bit sPAY=0, sACK=0, sSYN=0, sFIN=0;
217         /* Receive PayID, AckID and MSG */
218         short rPAY_ID=0, rACK_ID=0, rMSG=0;
219         /* Send PayID, AckID and MSG */
220         short sPAY_ID=0, sACK_ID=0, sMSG=0;
221
222         /* Temporary variables */
223         short fr_PAY_ID=0, fs_PAY_ID=0;
224         short timer=0, count=0;
225         bit want_close=0;
226
227         /* Handshake */
228
229         /* First phase of handshake */
230     listen:
231         printf("Server: LISTEN\n");
232         count=0;
233
234         do
235         /* Receive packet */
236         :: (nempty(in)) -> atomic {
237                 in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG;
238                 /* Simulation of 30 second waiting in this state */
239                 if
240                 :: (count>=MAX_ATTEMPTS) -> goto end_failure;
241                 :: else -> count++;
242                 fi;
243                 if
244                 :: (rSYN==1) -> {
245                         /* Store first received payload ID */
246                         fr_PAY_ID=rPAY_ID;
```

```
247                         goto respond;
248                     };
249                 /* Simulation of packet loss */
250                 :: (1) -> printf("Packet loss\n");;
251                 fi;
252             };
253         :: timeout -> goto end_failure;
254         od;
255
256         /* Second phase of handshake */
257     respond:
258         printf("Server: RESPOND\n");
259         sPAY=1; sSYN=1; sACK=1;
260         sMSG=rPAY_ID;
261         count=0;
262
263         /* Send response to the first packet */
264         out!sPAY,sACK,sSYN,sFIN,sPAY_ID,sACK_ID,sMSG;
265         /* Store ID of first sent payload packet */
266         fs_PAY_ID=sPAY_ID;
267
268         do
269         /* Receive packet */
270         :: (nempty(in)) -> atomic {
271                 in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG;
272                 if
273                 :: (count>=MAX_ATTEMPTS) -> goto end_failure;
274                 :: else -> count++;
275                 fi;
276                 if
277                 /* Still send response to the packet sent in REQUEST state */
278                 :: (rSYN==1 && rPAY_ID==fr_PAY_ID) -> {
279                         out!sPAY,sACK,sSYN,sFIN,sPAY_ID,sACK_ID,sMSG;
280                     };
281                 /* If received packet was sent from PARTOPEN state, then switch
282                  * to the OPEN state */
283                 :: (rACK==1 && rMSG==fs_PAY_ID) -> atomic {
284                         goto open;
285                     };
286                 /* Simulation of packet loss */
287                 :: (1) -> printf("Packet loss\n");
288                 fi;
289             };
290         :: timeout -> goto end_failure;
291         od;
292
293         /* End of handshake; normal communication */
294     open:
295         printf("Server: OPEN\n");
296         sSYN=0;
297         sMSG=rPAY_ID;
298         sPAY_ID = fs_PAY_ID+1;
299         timer=0;
300
301         /* Send response to the second packet */
302         out!sPAY,sACK,sSYN,sFIN,sPAY_ID,sACK_ID,sMSG;
303         sPAY_ID++;
304
305         do
306         :: timer++;
307             if
308             /* Send payload packet */
```

```
309             :: (nfull(out) && timer<TIMEOUT) -> atomic {
310                 out!sPAY,sACK,sSYN,sFIN,sPAY_ID,sACK_ID,sMSG;
311                 sPAY_ID++;
312                 };
313             /* Receive packet */
314             :: (nempty(in) && timer<TIMEOUT) -> atomic {
315                 in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG ->
316                 timer=0;
317                 do
318                 :: (rACK==1) -> {
319                         rACK=0;
320                         if
321                         :: (rMSG>=N) -> { goto closereq; };
322                         :: else -> skip;
323                         fi;
324                     };
325                 :: (rFIN==1) -> { rFIN=0; want_close=1; };
326                 :: else -> {
327                         if
328                         :: (want_close==1) -> { sMSG=rPAY_ID; goto closed; };
329                         :: else -> skip;
330                         fi;
331                         if
332                         :: (rPAY==1) -> {
333                                 sMSG=rPAY_ID;
334                                 sACK=1;
335                                 /* Send acknowledgment packet */
336                                 out!0,sACK,sSYN,sFIN,0,sACK_ID,sMSG;
337                                 sACK_ID++;
338                             };
339                         :: else -> skip;
340                         fi;
341                         break;
342                         };
343                 od;
344                 };
345             /* Simulation of packet loss */
346             :: (nempty(in) && timer<TIMEOUT) -> atomic {
347                 in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG ->
348                     printf("Packet loss\n");
349                 };
350             /* Other cases */
351             :: (empty(in) && full(out) && timer<TIMEOUT) -> skip;
352             /* No response from client */
353             :: (timer>=TIMEOUT) -> goto end_failure;
354             fi;
355         od;
356
357         /* Taerdown */
358     closereq:
359         printf("Server: CLOSEREQ\n");
360         sFIN=1;
361         count=0;
362         timer=0;
363
364         do
365         :: timer++;
366             if
367             /* Send payload packet */
368             :: (nfull(out) && timer<TIMEOUT) -> atomic {
369                 out!sPAY,sACK,sSYN,sFIN,sPAY_ID,sACK_ID,sMSG;
370                 sPAY_ID++;
```

```
371             };
372         /* Receive packet */
373         :: (nempty(in) && timer<TIMEOUT) -> atomic {
374             in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG ->
375             timer=0;
376             do
377             :: (rACK==1) -> { rACK=0; };
378             :: (rFIN==1) -> { rFIN=0; want_close=1; };
379             :: else -> {
380                     if
381                     :: (want_close==1) -> { sMSG=rPAY_ID; goto closed; };
382                     :: else -> skip;
383                     fi;
384                     if
385                     :: (rPAY==1) -> {
386                             sMSG=rPAY_ID;
387                             sACK=1;
388                             /* Send acknowledgment packet */
389                             out!0,sACK,sSYN,sFIN,0,sACK_ID,sMSG;
390                             sACK_ID++;
391                         };
392                     :: else -> skip;
393                     fi;
394                     break;
395                     };
396             od;
397             /* If client ignores FIN flag, then terminate connection */
398             if
399             :: (count>=MAX_ATTEMPTS) -> goto end_failure;
400             :: else count++;
401             fi;
402             };
403         /* Simulation of packet loss */
404         :: (nempty(in) && timer<TIMEOUT) -> atomic {
405             in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG ->
406                 printf("Packet loss\n");
407             };
408         /* Other cases */
409         :: (empty(in) && full(out) && timer<TIMEOUT) -> skip;
410         /* No response from client */
411         :: (timer>=TIMEOUT) -> goto end_failure;
412         fi;
413     od;
414
415 closed:
416     printf("Server: CLOSED\n");
417     sPAY=1; sACK=1; sSYN=0; sFIN=1;
418     count=0;
419
420     /* Send response to payload packet with FIN flag */
421     out!sPAY,sACK,sSYN,sFIN,sPAY_ID,sACK_ID,sMSG;
422
423     do
424     :: (nempty(in)) -> atomic {
425             in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG;
426             if
427             :: (count>=MAX_ATTEMPTS) -> goto end_failure;
428             :: else -> count++;
429             fi;
430             if
431             :: (rPAY==1 && rFIN==1) -> atomic {
432                     sMSG=rPAY_ID;
```

```
433                         out!sPAY,sACK,sSYN,sFIN,sPAY_ID,sACK_ID,sMSG;
434                     }
435                 /* Simulation of packet loss */
436                 :: (1) -> printf("Packet loss\n");
437                 fi;
438             };
439         :: timeout -> { goto end_success; };
440         od;
441
442 end_failure:
443         printf("Server: failure\n");
444         do
445         :: in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG;
446         :: timeout -> goto end;
447         od;
448
449 end_success:
450         printf("Server: success\n");
451         do
452         :: in?rPAY,rACK,rSYN,rFIN,rPAY_ID,rACK_ID,rMSG;
453         :: timeout -> goto end;
454         od;
455
456 end:
457         skip;
458 }
459
460 /* Main process */
461 init
462 {
463         /*                      PAY, ACK, SYN, FIN, PAY_ID ACK_ID MSG */
464         chan q1 = [QLEN] of {bit, bit, bit, bit, short, short, short};
465         chan q2 = [QLEN] of {bit, bit, bit, bit, short, short, short};
466
467         atomic {run Server(q1, q2); run Client(q2, q1);};
468 }
```

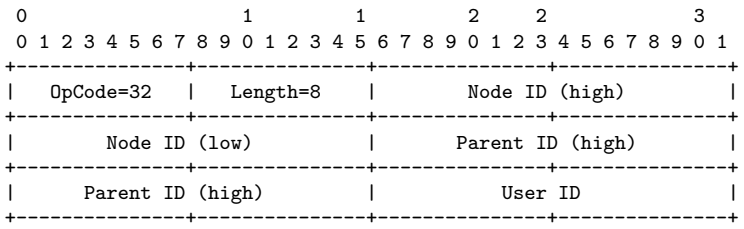
Výsledky verifikace komunikace na datagramovém spojení

```
$ spin -m -a  dgram.pml
$ gcc -o dgram -DBITSTATE -DSAFETY pan.c
$ ./dgram

Depth=    1028 States=    1e+06 Transitions= 1.52e+06 Memory=     2.539 t=   1.21 R=   8e+05
Depth=    1087 States=    2e+06 Transitions= 3.12e+06 Memory=     2.539 t=   2.46 R=   8e+05
Depth=    1087 States=    3e+06 Transitions= 4.87e+06 Memory=     2.539 t=   3.87 R=   8e+05

(Spin Version 5.2.5 -- 17 April 2010)
+ Partial Order Reduction

Bit statespace search for:
never claim          - (none specified)
assertion violations +
cycle checks        - (disabled by -DSAFETY)
invalid end states +

State-vector 124 byte, depth reached 1087, errors: 0
  3363590 states, stored
  2187890 states, matched
  5551480 transitions (= stored+matched)
  2253470 atomic steps
35231 lost messages

hash factor: 2.49394 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
  410.594 equivalent memory usage for states (stored*(State-vector + overhead))
    1.000 memory used for hash array (-w23)
    1.000 memory used for bit stack
    0.343 memory used for DFS stack (-m10000)
    2.539 total actual memory usage

unreached in proctype Client
(0 of 173 states)
unreached in proctype Server
line 313, state 102, "(1)"
(1 of 232 states)
unreached in proctype :init:
(0 of 4 states)

pan: elapsed time 4.46 seconds
pan: rate 754168.16 states/second
```

```
$ spin -m -a  dgram.pml
$ gcc -o dgram -DNP -DBITSTATE pan.c
$ ./dgram -l

Depth=    1195 States=    1e+06 Transitions= 1.87e+06 Memory=     1.598 t=   3.45 R=   3e+05
Depth=    1774 States=    2e+06 Transitions= 3.46e+06 Memory=     1.598 t=   6.97 R=   3e+05
Depth=    1774 States=    3e+06 Transitions= 5.21e+06 Memory=     1.598 t=   10.8 R=   3e+05

(Spin Version 5.2.5 -- 17 April 2010)
+ Partial Order Reduction

Bit statespace search for:
never claim          +
assertion violations + (if within scope of claim)
non-progress cycles  + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 128 byte, depth reached 1774, errors: 0
   370551 states, stored (3.47399e+06 visited)
  2654992 states, matched
  6128978 transitions (= visited+matched)
  2395153 atomic steps
37565 lost messages

hash factor: 2.41469 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
   50.887 equivalent memory usage for states (stored*(State-vector + overhead))
    1.000 memory used for hash array (-w23)
    0.038 memory used for bit stack
    0.305 memory used for DFS stack (-m10000)
    1.598 total actual memory usage

unreached in proctype Client
(0 of 173 states)
unreached in proctype Server
line 313, state 102, "(1)"
(1 of 232 states)
unreached in proctype :init:
(0 of 4 states)

pan: elapsed time 12.7 seconds
pan: rate 273326.99 states/second
```

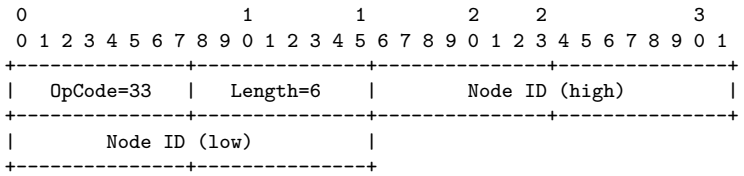
Kapitola 10

Struktury příkazů



Obrázek 10.1: Struktura příkazu pro vytvoření nového uzlu

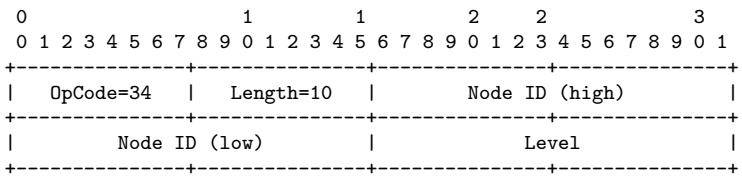
- *OpCode=32* - položka určující, že se jedná o příkaz *'Node Create'*
- *Length* - délka příkazu v bytech. Minimální délka je 8 bytů.
- *Node ID* - identifikátor nového uzlu.
- *Parent ID* - identifikátor nadřazeného uzlu.
- *User ID* - vlastník nového uzlu.



Obrázek 10.2: Struktura příkazu pro zrušení uzlu

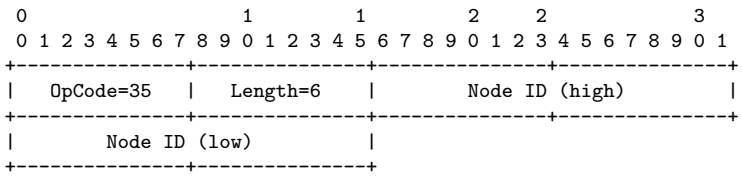
- *OpCode=33* - položka určující, že se jedná o příkaz *'Node Destroy'*

- *Length* - délka příkazu v bytech. Minimální délka je 6 bytů.
- *Node ID* - identifikátor uzlu, který se má zrušit



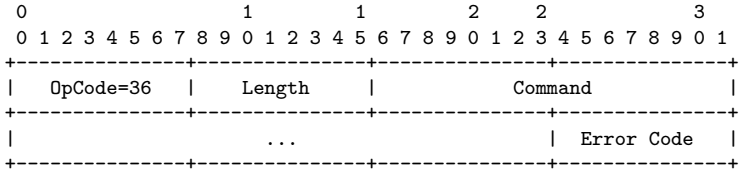
Obrázek 10.3: Struktura příkazu pro přihlášení se k uzlu

- *OpCode=34* - položka určující, že se jedná o příkaz *'Node Subscribe'*
- *Length* - délka příkazu v bytech. Minimální délka je 10 bytů.
- *Node ID* - identifikátor uzlu, kterému se chce klient přihlásit
- *Level* - počet úrovní přihlášení se k uzlům



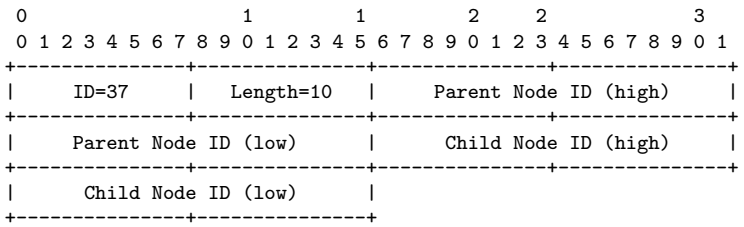
Obrázek 10.4: Struktura příkazu pro odhlášení se od uzlu

- *OpCode=35* - položka určující, že se jedná o příkaz *'Node UnSubscribe'*
- *Length* - délka příkazu v bytech. Minimální délka je 6 bytů.
- *Node ID* - identifikátor uzlu, od kterého se chce klient odhlásit (server klienta odhlásí od všech potomků daného uzlu)



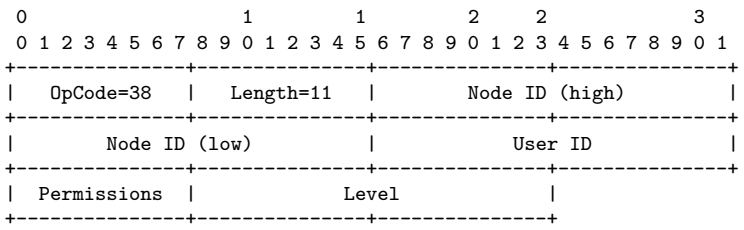
Obrázek 10.5: Struktura příkazu pro ohlášení chyby

- *OpCode=36* - položka určující, že se jedná o příkaz *'Cmd Error'*
- *Length* - délka příkazu v bytech.
- *Command* - obsah chybného příkazu, včetně položky *OpCode* a *Length*
- *Error Code* - kód chyby



Obrázek 10.6: Struktura příkazu pro změnu vazeb mezi uzly

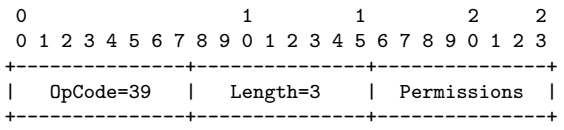
- *OpCode=37* - položka určující, že se jedná o příkaz *'Node Link'*
- *Length* - délka příkazu v bytech. Minimální délka je 10 bytů.
- *Parent Node ID* - uzel, který by měl být rodičem
- *Child Node ID* - uzel, který by měl být potomkem



Obrázek 10.7: Struktura příkazu pro změnu přístupových práv

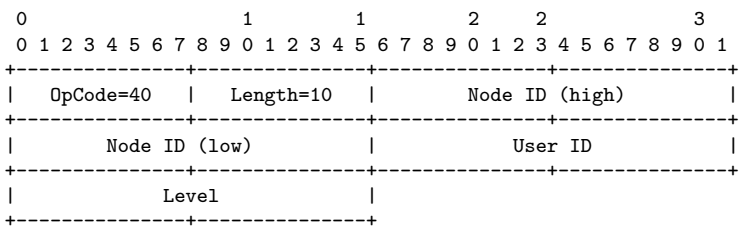
- *OpCode=38* - položka určující, že se jedná o příkaz *'Node Permission'*
- *Length* - délka příkazu v bytech. Minimální délka je 11 bytů.
- *Node ID* - uzel, který by měl být rodičem
- *User ID* - uzel, který by měl být potomkem

- *Permissions* - přístupová práva pro daného uživatele a uzel
- *Level* - počet rekurzivního nastavování daných práv



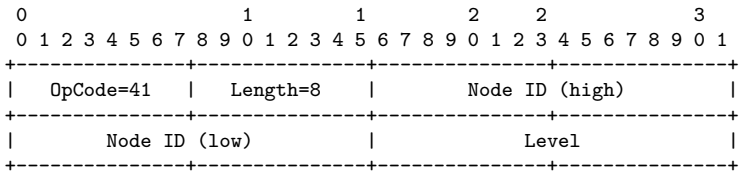
Obrázek 10.8: Struktura příkazu pro nastavení výchozích přístupových práv u ostatních uživatelů

- *OpCode=39* - položka určující, že se jedná o příkaz *'Node Default Permission'*
- *Length* - délka příkazu v bytech je vždy 3
- *Permissions* - defaultní přístupová práva pro nové uzly vytvořené klientem na daném spojení



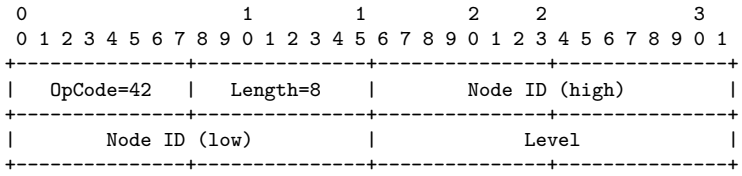
Obrázek 10.9: Struktura příkazu pro změnu vlastníka uzlu

- *OpCode=40* - položka určující, že se jedná o příkaz *'Node Owner'*
- *Length* - délka příkazu v bytech. Minimální délka je 10 bytů.
- *Node ID* - uzel, který by mít nového vlastníka
- *User ID* - identifikátor nového vlastníka
- *Level* - počet rekurzivního nastavování vlastníka



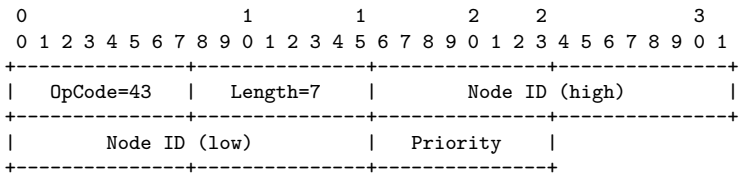
Obrázek 10.10: Struktura příkazu zamknutí uzlu

- *OpCode=41* - položka určující, že se jedná o příkaz *'Node Lock'*
- *Length* - délka příkazu v bytech. Minimální délka je 8 bytů.
- *Node ID* - uzel, který by měl být uzamčen
- *Level* - počet rekurzivního uzamčení uzlů



Obrázek 10.11: Struktura příkazu odemčení uzlu

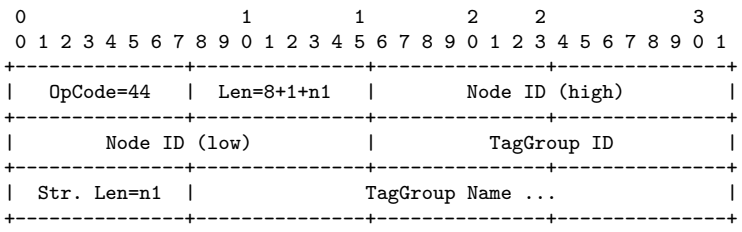
- *OpCode=42* - položka určující, že se jedná o příkaz *'Node UnLock'*
- *Length* - délka příkazu v bytech. Minimální délka je 8 bytů.
- *Node ID* - uzel, který by měl být odemčen
- *Level* - počet rekurzivního odemčení uzlů



Obrázek 10.12: Struktura příkazu pro nastavení priority uzlu

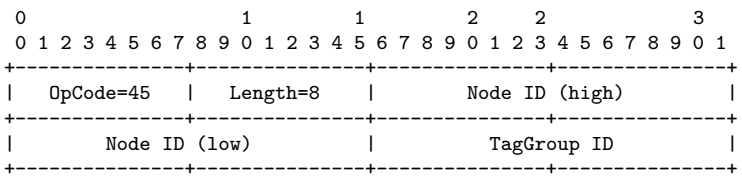
- *OpCode=43* - položka určující, že se jedná o příkaz *'Node Priority'*

- *Length* - délka příkazu v bytech. Minimální délka je 7 bytů.
- *Node ID* - rodičovský uzel, kterému má být nastavena priorita
- *Priotiry* - vlastní hodnota priority



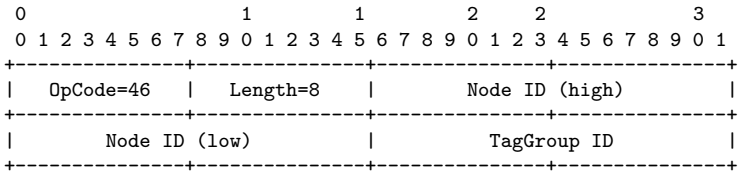
Obrázek 10.13: Struktura příkazu pro vytvoření skupiny tagů

- *OpCode=44* - položka určující, že se jedná o příkaz *'TagGroup Create'*
- *Length* - délka příkazu v bytech.
- *Node ID* - identifikátor uzlu, ve kterém má být vytvořena nová skupina tagů
- *TagGroup ID* - identifikátor skupiny tagů
- *String Length* - délka řetšzce se jménem skupiny tagů
- *TagGroup Name* - řetězec se jménem skupiny tagů



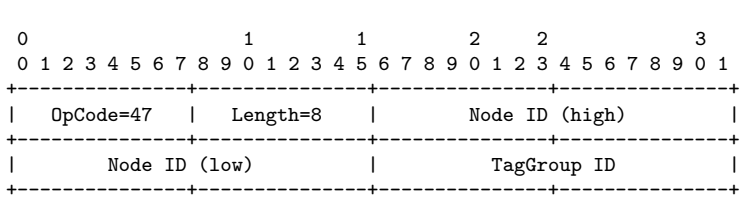
Obrázek 10.14: Struktura příkazu pro zrušení skupiny tagů

- *OpCode=45* - položka určující, že se jedná o příkaz *'TagGroup Destroy'*
- *Length* - délka příkazu v bytech. Minimální délka je 8 bytů.
- *Node ID* - identifikátor uzlu, ve kterém má být smazána skupina tagů
- *TagGroup ID* - identifikátor skupiny tagů, která má být smazána



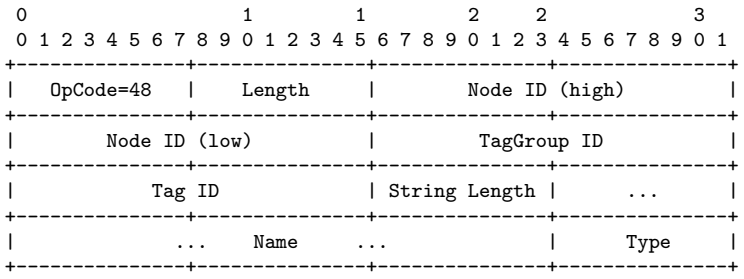
Obrázek 10.15: Struktura příkazu pro přihlášení se ke skupiny tagů

- *OpCode=46* - položka určující, že se jedná o příkaz *'TagGroup Subscribe'*
- *Length* - délka příkazu v bytech. Minimální délka je 8 bytů.
- *Node ID* - identifikátor uzlu, ve kterém se nachází skupina tagů
- *TagGroup ID* - identifikátor skupiny tagů, ke které se chce klient přihlásit



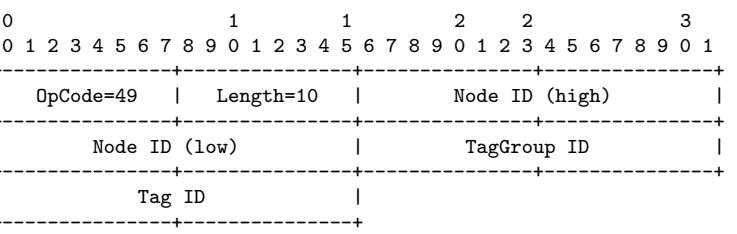
Obrázek 10.16: Struktura příkazu pro odhlášení se od skupiny tagů

- *OpCode=47* - položka určující, že se jedná o příkaz *'TagGroup Unsubscribe'*
- *Length* - délka příkazu v bytech. Minimální délka je 8 bytů.
- *Node ID* - identifikátor uzlu, ve kterém se nachází skupina tagů
- *TagGroup ID* - identifikátor skupiny tagů, od které se chce klient odhlásit



Obrázek 10.17: Struktura příkazu pro vytvoření nového tagu

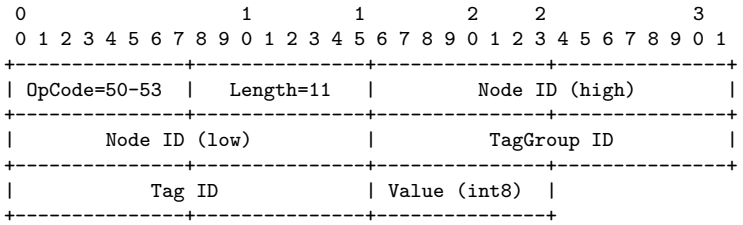
- *OpCode=48* - položka určující, že se jedná o příkaz *'Tag Create'*
- *Length* - délka příkazu v bytech.
- *Node ID* - identifikátor uzlu, ve kterém by měl být vytvořen nový tag
- *TagGroup ID* - identifikátor skupiny tagů, ve kterém by měl být vytvořen nový tag
- *Tag ID* - identifikátor nového tagu
- *String Length* - délka řetězce obsahující jméno tag
- *Name* - řetězec se jménem tagu
- *Type* - typ tagu (viz. strana 87)



Obrázek 10.18: Struktura příkazu pro zrušení tagu

- *OpCode=49* - položka určující, že se jedná o příkaz *'Tag Destroy'*
- *Length* - délka příkazu v bytech. Minimální délka je 10 bytů.
- *Node ID* - identifikátor uzlu, ve kterém má být tag zrušen
- *TagGroup ID* - identifikátor skupiny tagů, ve kterém má být tag zrušen

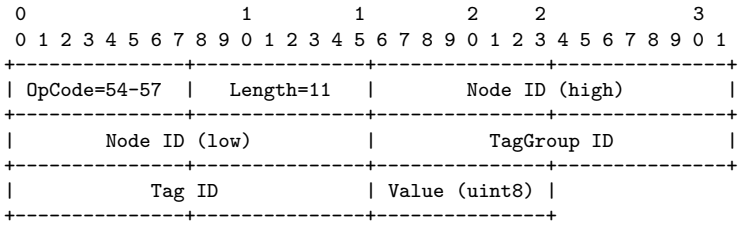
- *Tag ID* - identifikátor tagu, který má být zrušen



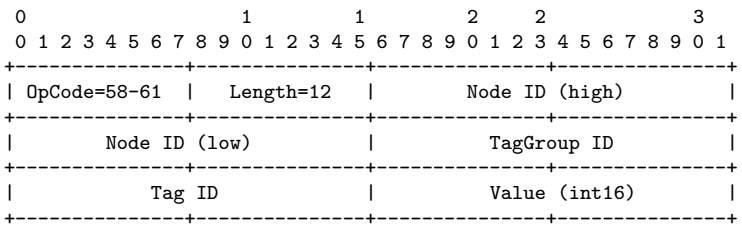
Obrázek 10.19: Struktura příkazu pro nastavení hodnoty tagu int8

- *OpCode=50-53* - položka určující, že se jedná o jednu z variant příkazu *'Tag Set (int8)'*
- *Length* - délka příkazu v bytech. Minimální délka je 11 bytů.
- *Node ID* - identifikátor uzlu, ve kterém má být nastavena hodnota tagu
- *TagGroup ID* - identifikátor skupiny tagů, ve které má být nastavena hodnota tagu
- *Tag ID* - identifikátor tagu, jehož hodnota má být nastavena
- *Value* - vlastní hodnota int8

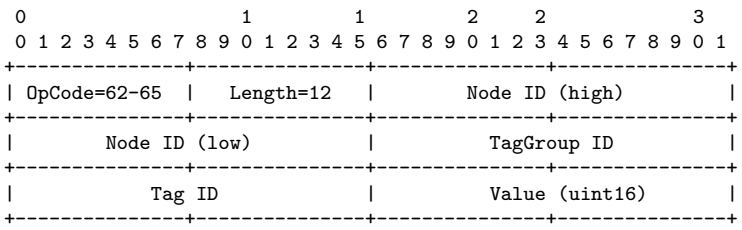
Poznámka: Následující příkazy *Tag Set* se liší pouze podle typu hodnoty, kterou nastavují, takže u nich nebude uváděn popis jednotlivých položek.



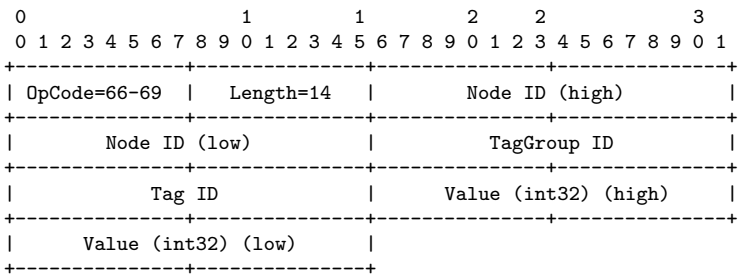
Obrázek 10.20: Struktura příkazu pro nastavení hodnoty tagu uint8



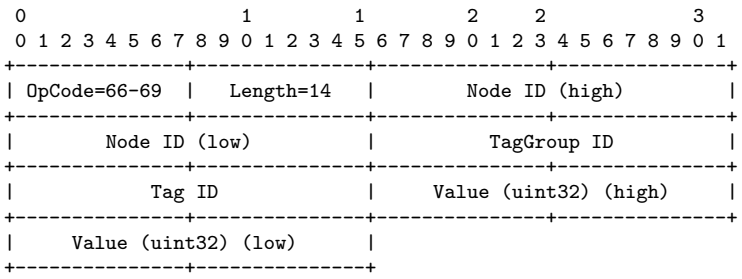
Obrázek 10.21: Struktura příkazu pro nastavení hodnoty tagu int16



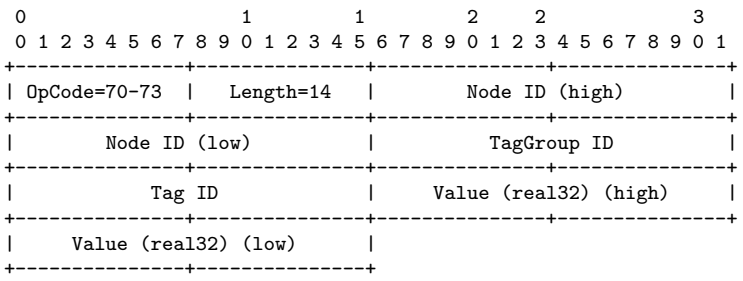
Obrázek 10.22: Struktura příkazu pro nastavení hodnoty tagu uint16



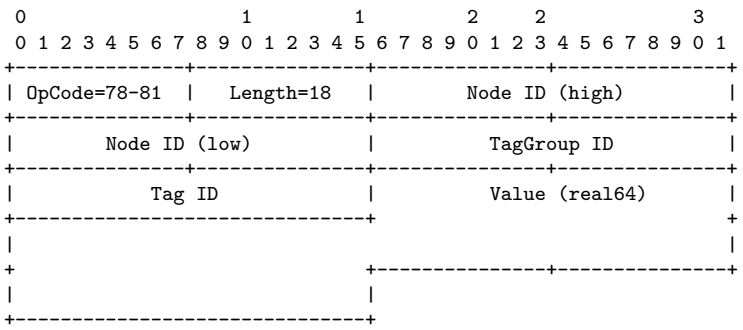
Obrázek 10.23: Struktura příkazu pro nastavení hodnoty tagu int32



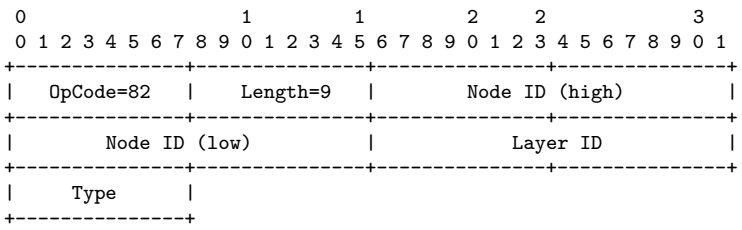
Obrázek 10.24: Struktura příkazu pro nastavení hodnoty tagu uint32



Obrázek 10.25: Struktura příkazu pro nastavení hodnoty tagu real32

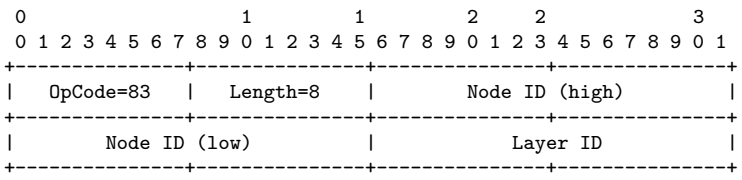


Obrázek 10.26: Struktura příkazu pro nastavení hodnoty tagu real64



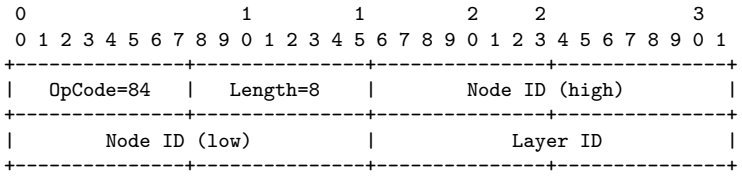
Obrázek 10.27: Struktura příkazu pro vytvoření vrstvy

- *OpCode=82* - položka určující, že se jedná o příkaz *'Layer Create'*
- *Length* - délka příkazu v bytech. Minimální délka je 9 bytů.
- *Node ID* - identifikátor uzlu, kde by měla být vytvořena nová vrstva
- *Layer ID* - identifikátor nové vrstvy (klient musí poslat hodnotu 0xFFFF)
- *Type* - typ vrstvy (viz. strana 91)



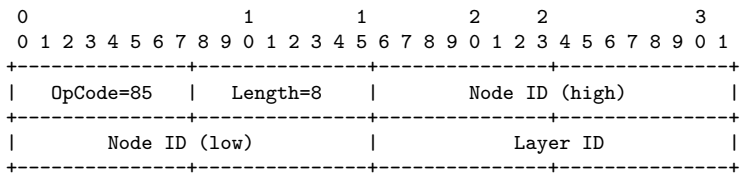
Obrázek 10.28: Struktura příkazu pro zrušení vrstvy

- *OpCode=83* - položka určující, že se jedná o příkaz *'Layer Destroy'*
- *Length* - délka příkazu v bytech. Minimální délka je 8 bytů.
- *Node ID* - identifikátor uzlu, ve kterém by měla být vrstva zrušena
- *Layer ID* - identifikátor vrstvy, která by měla být zrušena



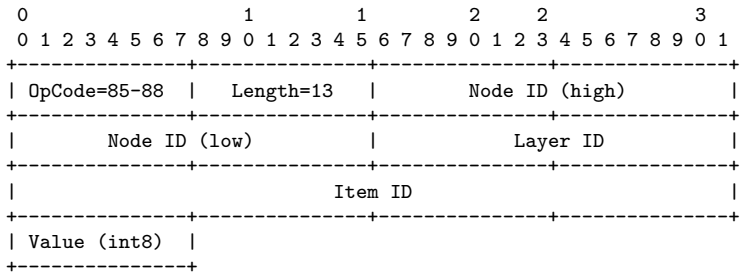
Obrázek 10.29: Struktura příkazu pro přihlášení se k vrstvě

- *OpCode=84* - položka určující, že se jedná o příkaz *'Layer Subscribe'*
- *Length* - délka příkazu v bytech. Minimální délka je 8 bytů.
- *Node ID* - identifikátor uzlu obsahující vrstvu, ke které se chce klient přihlásit
- *Layer ID* - identifikátor vrstvy, ke které se chce klient přihlásit



Obrázek 10.30: Struktura příkazu pro odhlášení se od vrstvy

- *OpCode=85* - položka určující, že se jedná o příkaz *'Layer Unsubscribe'*
- *Length* - délka příkazu v bytech. Minimální délka je 8 bytů.
- *Node ID* - identifikátor uzlu obsahující vrstvu, od které se chce klient odhlásit
- *Layer ID* - identifikátor vrstvy, od které se chce klient odhlásit

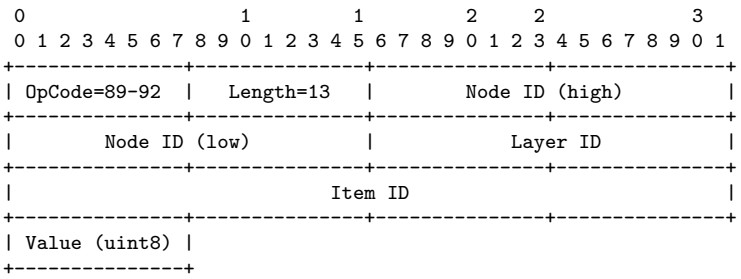


Obrázek 10.31: Struktura příkazu pro vytvoření/nastavení položky vrstvy, jejíž typ je int8

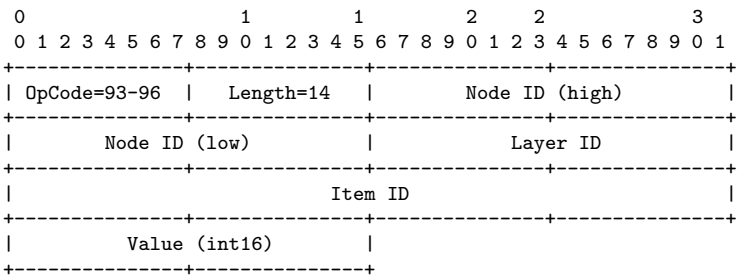
- *OpCode=85-88* - položka určující, že se jedná o jednu z variant příkazu *'Item Create'*
- *Length* - délka příkazu v bytech. Minimální délka je 13 bytů.

- *Node ID* - identifikátor uzlu, ve kterém má být vytvořena/nastavena položka vrstvy
- *Layer ID* - identifikátor vrstvy, ve které má být vytvořena/nastavena položka
- *Item ID* - identifikátor vlastní položky vrstvy
- *Value* - nová hodnota odkazované položky

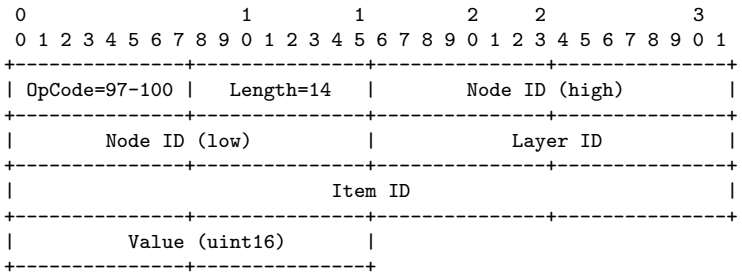
Poznámka: Následující příkazy *Item Set* se opět liší pouze podle typu hodnoty, kterou nastavují, takže u nich nebude uváděn popis jednotlivých položek.



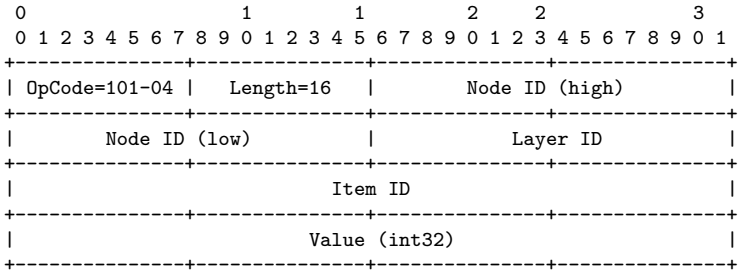
Obrázek 10.32: Struktura příkazu pro vytvoření/nastavení položky vrstvy, jejíž typ je uint8



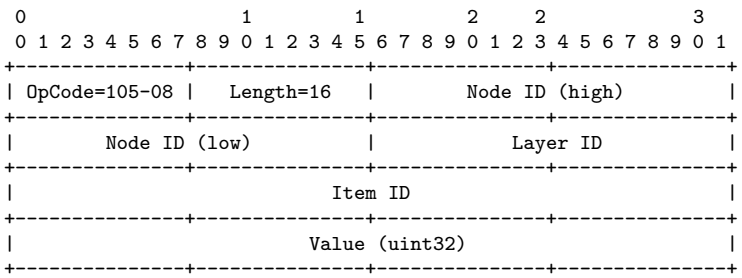
Obrázek 10.33: Struktura příkazu pro vytvoření/nastavení položky vrstvy, jejíž typ je int16



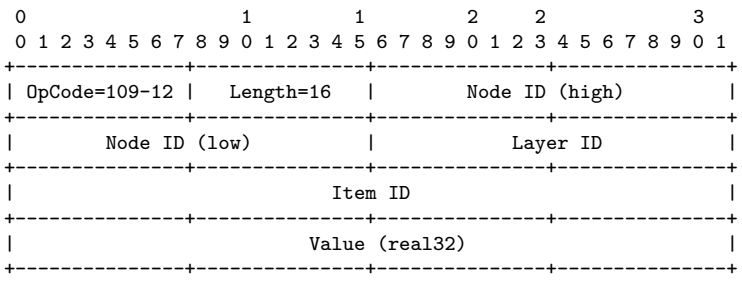
Obrázek 10.34: Struktura příkazu pro vytvoření/nastavení položky vrstvy, jejíž typ je uint16



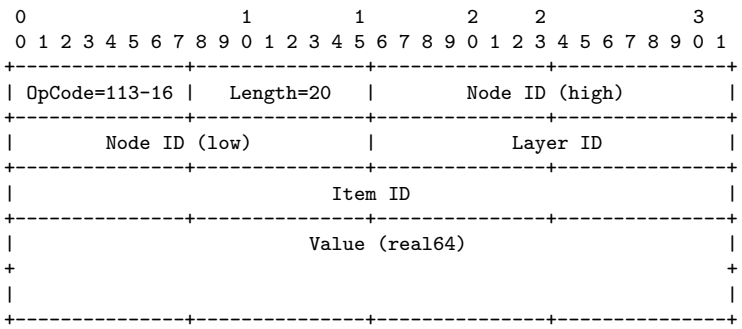
Obrázek 10.35: Struktura příkazu pro vytvoření/nastavení položky vrstvy, jejíž typ je int32



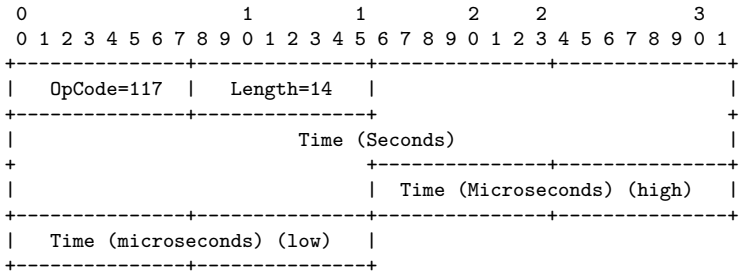
Obrázek 10.36: Struktura příkazu pro vytvoření/nastavení položky vrstvy, jejíž typ je uint32



Obrázek 10.37: Struktura příkazu pro vytvoření/nastavení položky vrstvy, jejíž typ je real32



Obrázek 10.38: Struktura příkazu pro vytvoření/nastavení položky vrstvy, jejíž typ je real64



Obrázek 10.39: Struktura příkazu pro nastavení hodnoty tagu real32

- *OpCode=117* - položka určující, že se jedná o jednu z variant příkazu *'Time Stamp'*
- *Length* - délka příkazu v bytech. Minimální délka je 14 bytů.
- *Time (seconds)* - UTC čas v sekundách od půlnoci 1. ledna 1970

- *Time (microseconds)* - počet mikrosekund od poslední ukončené sekundy

Kapitola 11

Verse API

V příloze je uvedený pouze výčet jednotlivých funkcí nového Verse API. Kompletní popis Verse API je dostupný v on-line podobě na adrese:
`http://www.nti.tul.cz/en/WikiUser:Jiri.Hnidek/New_Verse_API.en`

Verse API pro programovací jazyk C/C++

```
int verse_send_connect_request(const char *hostname,
     const char *service,
     const uint16 flags,
     uint32 *session_id);

void register_receive_user_authenticate(void (*func)(const uint32 session_id,
     const char *username,
     const uint8 auth_methods_count,
     const uint8 *methods));

int verse_send_user_authenticate(const uint32 session_id,
     const char *username,
     const uint8 auth_type,
     const uint8 data_length,
     const char *data);

void register_receive_connect_accept(void (*func)(const uint32 session_id,
      const uint32 user_id,
      const uint32 avatar_id));

int verse_send_connect_terminate(const uint32 session_id,
      const char error_num);

void register_receive_connect_terminate(void (*func)(const uint32 session_id,
      const uint8 error_num));

int verse_callback_update(void);

char *verse_strerror(const int error_num);

int verse_send_node_create(const uint32 session_id);

void register_receive_node_create((*func)(const uint32 session_id,
     const uint32 node_id,
     const uint32 parent_id,
     const uint16 user_id));
```

```
int verse_send_node_destroy(const uint32 session_id,
    const uint32 node_id);

void register_receive_node_destroy((*func)(const uint32 session_id,
    const uint32 node_id));

int verse_send_node_subscribe(const uint32 session_id,
    const uint32 node_id,
    const uint32 level);

int verse_send_node_unsubscribe(const uint32 session_id,
    const uint32 node_id);

int verse_send_node_priority(const uint32 session_id,
    const uint32 node_id,
    const uint8 priority);

void register_receive_node_priority((*func)(const uint32 session_id,
    const uint32 node_id,
    const uint8 priority));

int verse_send_link_set(const uint32 session_id,
    const uint32 parent_node_id,
    const uint32 child_node_id);

void register_receive_link_set((*func)(const uint32 session_id,
    const int16 link_id,
    const uint32 parent_node_id,
    const uint32 child_node_id));

int verse_send_node_grant_permissions(const uint32 session_id,
    const uint32 node_id,
    const uint16 user_id,
    const uint8 permissions);

void register_receive_node_grant_permissions((*func)(const uint32 session_id,
    const uint32 node_id,
    const uint16 user_id,
    const uint8 permissions));

int verse_send_set_node_owner(const uint32 session_id,
    const uint32 node_id,
    const uint16 user_id);

void register_receive_set_node_owner((*func)(const uint32 session_id,
    const uint32 node_id,
    const uint16 user_id));

int verse_send_set_node_umask(const uint32 session_id,
    const uint16 user_id,
    const uint8 permissions);

void register_receive_set_node_umask((*func)(const uint32 session_id,
    const uint16 user_id,
    const uint8 permissions));

int verse_send_node_lock(const uint32 session_id,
    const uint32 node_id);

void register_receive_node_lock((*func)(const uint32 session_id,
    const uint32 node_id));
```

```
int verse_send_node_unlock(const uint32 session_id,
    const uint32 node_id);

void register_receive_node_unlock((*func)(const uint32 session_id,
    const uint32 node_id));

int verse_send_tag_group_create(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id = 0xFF,
    const char *name);

void register_receive_tag_group_create((*func)(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id,
    const char *name));

int verse_send_tag_group_destroy(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id);

void register_receive_tag_group_destroy((*func)(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id);

int verse_send_tag_group_subscribe(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id);

int verse_send_tag_group_unsubscribe(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id);

int verse_send_tag_create(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id,
    const uint16 tag_id=0xFF,
    const char *name,
    const unit8 type);

void register_receive_tag_create((*func)(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id,
    const uint16 tag_id,
    const uint8 type,
    const char *name));

int verse_send_tag_int8(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id,
    const uint16 tag_id,
    const int8 value);

int verse_send_tag_uint8(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id,
    const uint16 tag_id,
    const uint8 value);

int verse_send_tag_int16(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id,
```

```
    const uint16 tag_id,
    const int16 value);

int verse_send_tag_uint16(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id,
    const uint16 tag_id,
    const uint16 value);

int verse_send_tag_int32(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id,
    const uint16 tag_id,
    const int32 value);

int verse_send_tag_uint32(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id,
    const uint16 tag_id,
    const uint32 value);

int verse_send_tag_real32(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id,
    const uint16 tag_id,
    const real32 value);

int verse_send_tag_real64(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id,
    const uint16 tag_id,
    const real64 value);

int verse_send_tag_string8(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id,
    const uint16 tag_id,
    const char *string);

int verse_send_tag_destroy(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id,
    const uint16 tag_id);

void register_receive_tag_destroy((*func)(const uint32 session_id,
    const uint32 node_id,
    const uint16 group_id,
    const uint16 tag_id));

int verse_send_layer_create(const uint32 session_id,
    const uint32 node_id,
    const uint16 layer_id,
    const unit8 type);

void register_receive_layer_create((*func)(const uint32 session_id,
    const uint32 node_id,
    const uint16 layer_id,
    const unit8 type));

int verse_send_layer_destroy(const uint32 session_id,
    const uint32 node_id,
    const uint16 layer_id);
```

```
void register_receive_layer_destroy((*func)(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id));

int verse_send_layer_subscribe(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id);

void verse_send_layer_unsubscribe(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id);

int verse_send_create_item_uint8(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const uint8 *value);

void register_receive_create_item_uint8((*func)(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const uint8 *value));

int verse_send_create_item_int8(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const int8 *value);

void register_receive_create_item_int8((*func)(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const int8 *value));

int verse_send_create_item_uint16(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const uint16 *value);

void register_receive_create_item_uint16((*func)(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const uint16 *value));

int verse_send_create_item_int16(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const int16 *value);
```

```
void register_receive_create_item_int16((*func)(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const int16 *value));

int verse_send_create_item_uint32(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const uint32 *value);

void register_receive_create_item_uint32((*func)(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const uint32 *value));

int verse_send_create_item_int32(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const int32 *value);

void register_receive_create_item_int32((*func)(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const int32 *value));

int verse_send_create_item_real32(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const real32 *value);

void register_receive_create_item_real32((*func)(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const real32 *value));

int verse_send_create_item_real64(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const real64 *value);

void register_receive_create_item_real64((*func)(const uint32 session_id,
     const uint32 node_id,
     const uint16 layer_id,
     const uint32 item_id,
     const uint16 num,
     const real64 *value));
```

```
int verse_send_destroy_item(const uint32 session_id,
    const uint32 node_id,
    const uint16 layer_id,
    const uint32 item_id);

void register_receive_destroy_item((*func)(const uint32 session_id,
    const uint32 node_id,
    const uint16 layer_id,
    const uint32 item_id));
```